ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής · Υπολογιστών
Εργαστήριο Λογικής · Επιστήμης Υπολογιστών

Αλγόριθμοι για Γραφήματα Εναλλακτικών Διαδρομών σε Οδικά Δίκτυα : Επισκόπηση και Αξιολόγηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νεφέλη Χαλαστάνη

Επιβλέπων : Δημήτρης Φωτάκης
Λέκτορας Ε.Μ.Π.

Αθήνα, Απρίλιος 2013

Στην οικογένειά μου

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής · Υπολογιστών
Εργαστήριο Λογικής · Επιστήμης Υπολογιστών

Αλγόριθμοι για Γραφήματα Εναλλακτικών Διαδρομών σε Οδικά Δίκτυα : Επισκόπηση και Αξιολόγηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νεφέλη Χαλαστάνη

Επιβλέπων : Δημήτρης Φωτάκης
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Απριλίου 2013

....
Δ. Φωτάκης
Λέκτορας Ε.Μ.Π.

....
Χ. Ζαρολιάγκης
Καθηγητής Παν. Πατρών

....
Σ. Ζάχος
Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2013

....

Νεφέλη Χαλαστάνη

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Contents

ABSTRACT

Nowadays, mobility has gained great importance. People travel daily either for their works or for pleasure making route planning a significant aspect of their everyday life. Although first algorithms solving this kind of problem, such as Dijkstra and Bellman-Ford, are quite old, the new circumstances require more efficient solutions taking into account various parameters related to driver's preferences, environmental conditions, financial conditions, traffic congestions and many other aspects.

Under these new requirements and needs, algorithmic engineering exhibited an impressive surge of interest during the last years in the field of route planning, developing new algorithmic approaches that solve more sophisticated variants of the shortest path problem. For example, researchers developed speed-up techniques computing queries in large-scale networks with great time efficiency, introduced new concepts of multi-modal and multi-objective routing etc.

In this thesis, we focus on a variant of the shortest path problem that aims to find multiple alternative routes from source to target, apart from the single shortest path. Research on this field is very important since finding a set of alternatives can be useful for various reasons. For example, the need to avoid costly roads, the increasing occurrence of traffic congestions, driver's preferences to avoid some streets are cases that a single route would not be enough.

So, we present plateau method which is a significant approach on this field of interest based on paths with constant distances from source and target, along them (plateaux). We analyze step by step its stages, we understand the methodology and we implement it in our own C++ program. We further combine our plateau approach with penalty algorithm which, as its name states, penalizes the edges already belonging in an alternative. Apart from the two interesting implementations, we aim to evaluate the resulting alternative routes so as to choose the best ones. The superiority of each single alternative is examined on the basis of specific criteria we introduce. These criteria check the quality of the candidate plateaux, the amount of disjointness between the paths, the average length, the local optimality. These criteria are incorporated in the C++ implementations. Last but not least, in order to examine the efficiency of these algorithms, we conduct an experimental study for various values of the parameters in large-scale road networks, we export statistical data and we draw conclusions.

ΠΕΡΙΛΗΨΗ

Στις μέρες μας, οι μετακινήσεις έχουν αποκτήσει μεγάλη σημασία. Οι άνθρωποι ταξιδεύουν καθημερινά είτε για τις δουλειές τους είτε για λόγους αναψυχής με αποτέλεσμα οι μετακινήσεις να αποτελούν ένα σημαντικό μέρος της ζωής τους. Αν και οι πρώτοι αλγόριθμοι, που έλυναν τέτοιου είδους προβλήματα, όπως ο Dijkstra και ο Bellman-Ford είναι αρκετά παλιοί, οι νέες συνθήκες απαιτούν πιο αποδοτικές λύσεις λαμβάνοντας ταυτόχρονα υπόψη ποικίλες παραμέτρους όπως τις προτιμήσεις των οδηγών, τις περιβαλλοντικές συνθήκες, τις οικονομικές συγκυρίες, την κυκλοφοριακή συμφόρηση και πολλά άλλα.

Κάτω από αυτές τις νέες απαιτήσεις, το algorithmic engineering δείχνει τα τελευταία χρόνια μεγάλο ενδιαφέρον σχετικά με την εύρεση και το σχεδιασμό διαδρομών, αναπτύσσοντας νέες αλγοριθμικές μεθόδους οι οποίες λύνουν πιο εξεζητημένες εκδοχές του προβλήματος εύρεσης βέλτιστης διαδρομής. Για παράδειγμα, ερευνητές έχουν αναπτύξει τεχνικές επιτάχυνσης του Dijkstra με τις οποίες καταφέρνουν να υπολογίσουν ερωτήματα μεταξύ αφετηρίας-προορισμού σε μεγάλης κλίμακας δίκτυα με πολύ μικρή χρονική πολυπλοκότητα ή έχουν εισάγει την έννοια εύρεσης διαδρομών με συνδυασμό των μέσων μαζικής μεταφοράς κλπ.

Σε αυτή τη διπλωματική εργασία, επικεντρωνόμαστε σε μία παραλλαγή του προβλήματος εύρεσης συντομότερης διαδρομής η οποία έχει στόχο να εντοπίζει πολλαπλές εναλλακτικές διαδρομές από μία αφετηρία σε έναν προορισμό, πέραν της βέλτιστης. Η έρευνα πάνω σε αυτό το αντικείμενο είναι ιδιαίτερα σημαντική μιας και ο υπολογισμός ενός συνόλου εναλλακτικών διαδρομών είναι πολύ χρήσιμος σε πληθώρα περιπτώσεων. Για παράδειγμα, η αποφυγή διοδίων, η αυξανόμενη εμφάνιση κυκλοφοριακών προβλημάτων, η προτίμηση ενός οδηγού να αποφεύγει κακόφημους δρόμους αποτελούν λίγες περιπτώσεις όπου ο υπολογισμός μιας μοναδικής διαδρομής δεν επαρκεί.

Συνεπώς, στην εργασία αυτή, παρουσιάζουμε τη μέθοδο του plateau (plateau method) η οποία αποτελεί μία σημαντική προσέγγιση σε αυτό το πεδίο ενδιαφέροντος και βασίζεται στην εύρεση μονοπατιών οι κόμβοι των οποίων έχουν σταθερές αποστάσεις από την αφετηρία και τον προορισμό (plateaux). Αναλύουμε βήμα προς βήμα όλα τα στάδια του αλγορίθμου, κατανοούμε τη μεθοδολογία και κάνουμε τη δική μας υλοποίηση σε C++. Στη συνέχεια, συνδυάζουμε το plateau με τον αλγόριθμο penalty ο οποίος, όπως δηλώνει και το όνομά του, εισάγει κάποια ποινή (= επιπλέον βάρος) στις ακμές των εναλλακτικών που ήδη έχουν υπολογιστεί. Εκτός από αυτές τις δύο υλοποιήσεις, στόχος μας επίσης είναι να αξιολογήσουμε τις υποψήφιες εναλλακτικές που προκύπτουν ώστε να διαλέξουμε τις καλύτερες. Η υπεροχή της κάθε εναλλακτικής εξετάζεται με βάση κάποια συγκεκριμένα κριτήρια. Αυτά τα κριτήρια εξετάζουν την ποιότητα των plateaux, το ποσοστό των κοινών τμημάτων των εναλλακτικών, το μέσο μήκος τους συγκριτικά με το βέλτιστο, την τοπική βελτιστότητα. Αυτά τα κριτήρια επίσης ενσωματώνονται στις υλοποιήσεις μας σε C++. Τέλος, για να εξετάσουμε την αποδοτικότητα αυτών των αλγορίθμων, εκτελούμε πειράματα για διάφορες τιμές των παραμέτρων, σε οδικά δίκτυα μεγάλης κλίμακας, εξάγουμε στατιστικά αποτελέσματα και βγάζουμε συμπεράσματα.

ΕΥΧΑΡΙΣΤΙΕΣ

Κατ' αρχάς, θα ήθελα να ευχαριστήσω τον καθηγητή και επιβλέποντα της διπλωματικής μου, κ. Δημήτρη Φωτάκη. Με τις γνώσεις, τον ενθουσιασμό, τη μεταδοτικότητά του και τις εξαιρετικές διαλέξεις του, με έκανε να αγαπήσω τους αλγορίθμους και να αναλάβω μία διπλωματική σε αυτό το ερευνητικό αντικείμενο. Θα ήθελα επίσης να τον ευχαριστήσω για την εμπιστοσύνη που μου έδειξε καθώς και για το χρόνο που πέρασε μαζί μου συζητώντας για το θέμα και λύνοντας τις απορίες μου.

Θα ήθελα επίσης να ευχαριστήσω τον καθηγητή κ. Χρήστο Ζαρολιάγκη, από το Πανεπιστήμιο της Πάτρας, ο οποίος μου έδωσε τη δυνατότητα να ασχοληθώ με αυτό το πολύ ενδιαφέρον θέμα και μου παρείχε μία έτοιμη βιβλιοθήκη σε C++, την οποία ανέπτυξε με την ερευνητική του ομάδα, και η οποία περιέχει έτοιμες και αποδοτικές δομές δεδομένων και υλοποιήσεις αλγορίθμων, που κρίθηκαν απαραίτητες για την εκπόνηση της διπλωματικής μου.

Επιπλέον, θα ήθελα να ευχαριστήσω τον Παναγιώτη Μιχαήλ, μεταπτυχιακό φοιτητή στην ερευνητική ομάδα του κ. Ζαρολιάγκη ο οποίος με βοήθησε για το στήσιμο του περιβάλλοντος και ήταν πάντα πρόθυμος να απαντήσει στις απορίες μου.

Ένα μεγάλο ευχαριστώ θα ήθελα να δώσω στην οικογένειά μου μου η οποία με στήριξε καθ' όλη τη διάρκεια της φοιτητικής μου ζωής και χωρίς αυτούς θα ήταν αδύνατο να ολοκληρώσω τις σπουδές μου.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους μου για τη στήριξή τους. Αποτέλεσαν ένα σημαντικό κομμάτι της φοιτητικής μου ζωής.

Νεφέλη Χαλαστάνη

Απρίλιος 2013

CHAPTER 1

INTRODUCTION

1.1 Motivation and problem statement

Mobility is very important in our society. People live in one city and work in another. They visit friends and family living in different parts of the country. Even leisure time is not always spent in their residence. Consequently, finding best possible routes in transport networks from a given source to a given target location becomes an everyday problem. Many people daily deal with this question when planning trips either with their cars or with public transportation.

First algorithms to solve this problem are quite old and are presented in the 1950s and 60s by Dijkstra [Dij59], Bellmann and Ford [Jr.56, Bel58] and Hart, Nilsson and Raphael [HNR68] (A*). Since then, an impressive amount of work has been done to solve different shortest path cases.

Lately, the route planning problem has received considerable attention and has become one of the showpieces of real-world applications of algorithmics. Many reasons lead to this kind of interest :

- The large increase in transportations
- The increasing use of private vehicles in combination with CO_2 emissions and energy consumption that require more eco-friendly approaches in order to reduce the environmental footprint
- The increasing occurrence of traffic congestions that leads to alternative routes search
- The large real-world graphs like continental road networks that require more efficient computations
- The new financial circumstances that create the need to avoid costly roads and set new criteria in route planning

The above mentioned reasons lead to the development of new algorithmic approaches that solve more sophisticated variants of the shortest path problem.

In this thesis, we focus on the problem of finding efficiently alternative routes in road networks. Taking into account that every human likes choices and that today's fast route planning

algorithms usually compute just a single route, we consider a generalization of the well-known shortest path problem, in which not one but several alternative paths must be produced. Often, there exist several noticeable different paths from start to end which are almost optimal with respect to travel time. For a human, it is advantageous to be able to choose a route for his tour among a set of good alternatives. He may have personal preferences, knowledge for some routes, and bias against others which are unknown or difficult to obtain (e.g. slippery road). Also, routes can vary in different attributes besides travel time, for example in toll pricing, scenic value, fuel consumption or risk of traffic jams. The trade-off between those attributes depends on the person and is difficult to determine. By computing a set of good alternatives, the person himself can choose the route which best fits his needs.

So far, beginnings to compute alternative routes have been made, but this topic has not been studied thoroughly. We fill in this gap by describing mathematical definitions for such routes, introducing heuristics, implementing and combining methods to compute them and using new data structures that improve data locality and accelerate queries computation.

1.2 Related applications of shortest path algorithms

Routing is a widely researched topic in computer science, mainly because of its relevance to real world applications. Many software companies develop state of the art applications and web-services relating to routing and great projects on this field take place under the joint participation of universities, companies and organizations.

Navigation Systems . Car navigation systems are being offered as a special feature of new cars in an increasing number of car-brands. These car navigation systems are capable of taking over some of the tasks that are performed by the driver such as reading the map and determining the best route to the destination. A navigation system offers the driver the possibility to be guided to his destination, by means of spoken and visual advices. In order to achieve this, the driver first has to enter his

Figure 1.1: Built-in car navigation system

destination into the system. Such a destination may be a city center, an entire street, or an address including a house number. They should also take daily congestion patterns into account. Because a car navigation system uses a built-in computer to determine a route, it can compare many different routes and the user expects the system to determine the best possible or optimum route fast.

Web mapping services . Many software companies provide web mapping service applications and technology that power many map-based functionalities including routing and navigation. They offer street maps, route planners for travelling on foot, by car or public transport. Specifically, those services are computer software programmes, designed to plan a (optimal) route between two geographical locations using a journey planning engine, typically specialized for road networks as a road route planner. It can typically provide a list of places one will pass by, with crossroads and directions that must be followed, road numbers, distances, etc. Also, it usually provides an interactive map with a suggested route marked on it. Many online mapping sites offer road route planning like C2Logix, ViaMichelin, Google Maps, Bing Maps & Directions, Mapquest, Intermodal Journey Planner and TomTom Route Planner systems. Although as a route planning software is prone to mistakes if you try to get directions from destination A to B, the use of common sense is also required. Applications can typically also calculate the journey time and cost, some also display points of interest along the route.

Tourists trip planning applications . Many tourists visit a region or a city for one or more days and have to make a selection of the most valuable Points of Interest (POIs). This personal selection is based on information found on web sites, in articles, in magazines or in guidebooks from specialized book stores or libraries. Once the selection is made, the tourist decides on a route, keeping in mind the opening hours of the POIs and the available time. However, this procedure faces several difficulties and requires great organization and good combination and confirmation of the information acquired, since sometimes it may be out of date. For these reasons, web-based decision support applications have been developed and become excellent aid for tourists who want real life support for tourist planning problems. Based on an interest profile, up-to-date POI information and trip information, a (near-) optimal and feasible selection of POIs and a route between them can be suggested solving a generalization of the well-known travelling salesman problem. Similar functionalities for visiting POIs are being embedded in smartphones platforms using android and ios technologies.

Fleet management functionalities . A fleet route, also called Vehicle Routing Problem (generalization of TSP) is similar to a point-to-point route with VIA points, but with one important difference: every point visited had a load, and each vehicle has a maximum capacity, so one vehicle might not be enough to visit all VIA points. This means that several vehicles are usually needed, which adds a lot of complexity to the route calculations. Libraries (like Xtreme Route library) contain functionality for solving fleet routes, with several configurable parameters as vehicle capacity, VIA point load, depot, route type (round-trip, inbound, outbound) etc. As with point-to-point routing there are a lot of use cases for fleet routes, some of them are:

- Package delivery involving many or a fixed number of vehicles (if only 1 vehicle, the problem becomes a point-to-point route with VIA points)
- Transport of students to and from school
- Mail delivery
- Timber transportation

- Waste management

eCOMPASS Project . eCOMPASS introduces new mobility concepts and establishes a methodological framework for route planning optimization aiming at reducing the environmental impact of urban mobility. eCOMPASS aims at delivering a comprehensive set of tools and services for end users to enable eco-awareness in urban multi-modal transportations. eCOMPASS involves a generic architecture that will consider all types and scenarios of human and goods mobility in urban environments minimizing their
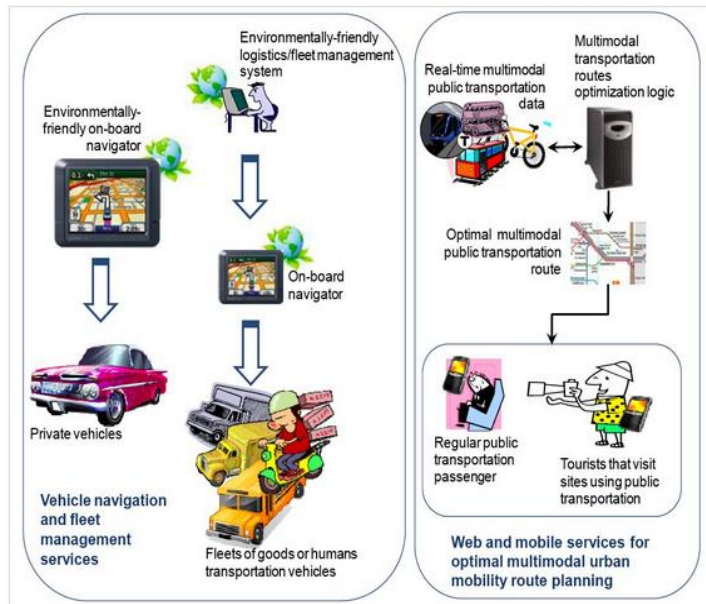
Figure 1.2 : eCOMPASS expected results from eCompass site

corresponding environmental impact. Firstly, the project will focus on the design and development of intelligent on-board and centralized vehicles' fleet management systems; the fundamental objective of eco-awareness will be addressed through employing intelligent traffic prediction and traffic balancing methods, while also taking into account driving behaviour and considering the option of car drivers transferred to means of public transportation at suitable locations. Secondly, eCOMPASS will develop web and mobile services providing multi-modal public transportation route planning, taking into account contextual information (such as location and time) as well as various restrictions and/or user constraints. Recommended routes will be optimized mainly in terms of the transports' environmental footprint, although additional objectives will also be considered. An important objective of eCOMPASS is to develop novel algorithmic solutions and deliver the respective services to familiar end-user mobile devices.

To sum up, there are numerous applications and projects relevant to shortest path problems that range from planning a motorcycle tour with a mobile device to facility location problems of large industrial companies. Although we mainly associate shortest path algorithms with routing and navigation problems in transportation, there are also major fields of science where these algorithms are widely applied. Specifically, two typical cases are :

Computer networks . Routing in computer networks is an essential functionality, which influences the network management and the quality of services in global networks. The management of the traffic flows has to satisfy requirements for volume of traffic to be transmitted, for avoidance of congestions and for decreasing the transmission delays. The optimal traffic management is a key issue for the quality of the information services. Routing in networks and applying shortest path algorithms is widely used in communication protocols in

WAN. The routing algorithm is described as network layer protocol (i.e. OSFP, BGP) that guides packets (information stored as small strings of bits) through the communication subset to their correct destinations.

Robotics . Due to the development of automation industry, robots are needed in more and more complex and dynamic environments. For example, the research of unmanned vehicle robots is increased, especially in military field. An important issue to be considered is how these robots move and avoid static or moving obstacles in the workplace. They need to implement both path tracking and obstacle avoidance algorithms that are borrowed or inspired from shortest path problems in graphs. They need to plan a path from a current to a goal position avoiding hurdles and in order to achieve that use Dijkstra and A* variants.

1.3 Contribution

In this thesis we approach the problem of finding efficiently alternative routes in large-scale road networks. Our main contribution is divided into three parts : a) reviewing basic algorithms on the field of alternative routes search, b) presenting analytically plateau method and criteria for evaluating method's results and c) conducting an experimental study on large-scale road networks using our practical implementations for plateau method and for a combination of plateau with penalty method.

Algorithms for finding alternative routes. As we stated above, this thesis focuses on the problem of finding alternative routes from source to target apart from the shortest one. So, it is necessary to study the algorithms already developed in this field. Specifically, we review some of the most important methods used for alternative route search, namely k-shortest paths, disjoint paths, pareto-optimal paths and penalty method, we describe the steps of each algorithm and we state their main drawbacks which led to the development of new approaches. Thanks to this study, the reader gets familiar with the field of interest and obtains an insight on the various methods developed.

Plateau Method and Criteria for evaluating results. The main part of our work is the study of Plateau method which is a significant algorithm for generating a plurality of diverse routes. So our main contribution lies in the analysis of the algorithm's stages which helps us to understand the methodology and implement it in our own C++ program. We further combine plateau algorithm with penalty algorithm (hybrid approach). Apart from the two interesting implementations, we evaluate the resulting alternative routes so as to choose the best ones. The superiority of each single alternative is examined on the basis of specific criteria we introduce. These criteria check the quality of the candidate plateaux, the amount of disjointness between the paths, the average length, the local optimality. So the reader understands Plateau method in depth and obtains a set of useful metrics to evaluate the results.

Experimental Study. Last but not least, we conduct an experimental study so as to examine the efficiency of Plateau method and the hybrid approach. We run several experiments in real large-scale road networks, we evaluate the results regarding the quality of the alternative routes found based on the criteria we defined and we further examine the time performance and memory usage of the algorithms. So, thanks to the experimental results and statistical data, the reader obtains a practical overview of the algorithms quality and their efficiency in real-world maps.

1.4 Overview

This thesis is organized as follows :

Chapter 2 : Definitions and related work. Chapter 2 lays the essential foundations for our work. We define basic notion for graphs including edge weights and paths. Moreover, we present the data structures used for graph representation and we make a comparison between them, necessary for the experimental results given in chapter 6. Furthermore, we model road networks as graphs and we introduce the concept of basic routing. Besides, we analyze basic improvements and speed-up techniques of Dijkstra algorithm as bidirectional Dijkstra, A*, Arc Flags, Highway Hierarchies, Contraction Hierarchies. Although these techniques are not directly associated with our main subject of interest, which is alternative routes in large-road networks, they provide us with a general theoretical background that is widely used in large-scale transportation and might easily be applied as a future work in the current thesis.

Chapter 3 : Alternative route algorithms and alternative graphs. Chapter 3 introduces us to the main subject of this thesis and several initial approaches for finding alternative routes are presented. First of all, we describe the alternative route graph that is given as a result from each algorithm's execution and then we describe the algorithms and theirs steps for computing more than one routes in a road network. As their name states, k-shortest paths compute the k shortest paths as alternative routes and regard sup-optimal paths. The computation of disjoint paths is similar, except that the paths must not overlap. Another approach uses several edge weights to compute Pareto-optimal paths. Given a set of weights, a path is called Pareto-optimal if it is better than any other paths for respectively at least one criterion. All Pareto-optimal paths can be computed by a generalized Dijkstra's algorithm. The penalty method iteratively computes shortest paths in the graph while increasing certain edge weights.

Chapter 4 : Plateau method and admissibility criteria. While the methods in chapter 3 are widely used in many applications until now, their results can suffer a lot, as they can be very long or very common etc. and as a result may not be admissible to the user. To avoid these problems, in chapter 4, we introduce the concept of admissible alternative routes. In other words, we define the properties that a good alternative must satisfy and we propose ways of measuring these properties in real-world maps. Specifically, alternative candidates should be

nonoverlapping to a large extend, locally optimal and not significantly larger than the optimal route.

Having defined these properties, in the second part of this chapter, we present plateau method or choice routing algorithm for finding alternative routes. The alternatives, found by plateau method, "naturally" meet the admissibility criteria to a greater extent (due to the way they are computed) than the methods presented in chapter 3. Furthermore, we combine plateau with the aforementioned penalty method (hybrid approach) in order to achieve better results. These two methods are going to be implemented and, thus, presented thoroughly in the next chapter.

Chapter 5 : Practical implementation and methodology. This chapter analyzes plateau method and hybrid method, presented in chapter 4, in a more practical way with references in our practical implementations. Specifically, we present our implementation strategy for plateau method using a block diagram. In this diagram, the stages of the algorithm are presented in the order they are meet in our programs. After that, each stage is described in detail. Last but not least, we present the basic methodology for the hybrid method, namely plateau method with penalization of edges.

Chapter 6 : Experiments and results. In Chapter 6, we conduct several alternative routes experiments. At first, we describe our experimental setup including the inputs used throughout the experiments (DIMACS10 maps), the experimental environment etc. In the next sections, we compare execution times for the different stages of our approaches, we show the mean targetFunction values for plateau and the hybrid implementation (combination of plateau and penalty) and we draw conclusions regarding the efficiency of the different approaches.

Chapter 7: Conclusion and outlook. Chapter 7 gives a final conclusion and an outlook regarding future research on the topic of alternative routes and specifically as an extension of the current thesis, taking into account various aspects, such as time dependency, real-time traffic etc.

CHAPTER 2

DEFINITIONS AND RELATED WORK

In section 2.1 of this chapter, we define the basic notion for graphs, while in section 2.2, we present some data structures that are used for the graph storage which exhibit significant characteristics ([MMPZ12]). Furthermore, we model road networks as graphs and we introduce the concept of basic routing. Besides, we analyze basic improvements and speed-up techniques of Dijkstra algorithm providing a general theoretical background widely-used in large-scale transportation.

2.1 Graph theory

In this section, we develop the basic notation which is needed throughout this thesis. Since all of our algorithms work on graphs, the underlying concepts of graph theory are introduced first.

Graphs . A graph $G = (V, E)$ is a tuple consisting of a finite set V of nodes and a set of $E \subseteq V \times V$ of edges. We say there is an edge from $u \in V$ to $v \in V$, if and only if $(u, v) \in E$. Usually $n = |V|$ denotes the number of nodes and $m = |E|$ denotes the number of edges. The graph obtained by flipping all edges is called backward graph $\overleftarrow{G} := (V, \overleftarrow{E})$ where $(u, v) \in \overleftarrow{E} \Longleftrightarrow (v, u) \in E$. Furthermore, a graph has a weight function $w : E \to \mathbb{R}^+$ that assigns a positive weight to each edge in G. For an edge $(v, w) \in E$, we usually write $w(v, w)$ instead of $w((v, w))$.

Edge Weights . The edge weights may indicate travel time, distance, or may be a combination of different parameters. The values of edge weights differentiate between time-independent and time-dependent route planning. Whereas for time-independent route planning it is sufficient to have constant weights, this concept is generalized to periodic functions to accommodate for different edge weights at different times of day.

Paths . A path P in G is a sequence of nodes

$$\langle u_1, \dots, u_k \rangle \quad \text{with } (u_i, u_{i+1}) \in E, \text{ for all } i = 1, \dots, k-1$$

For $1 \leq i \leq j \leq k$ the subpath of P between u_i and u_j is denoted $P' \subset P$ and it is a path itself which is fully contained in P. A path that passes via a node v is denoted by P_v.
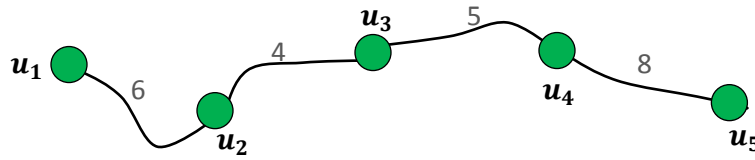
Figure 2.1 : A graph G with 5 nodes (u_1, u_2, u_3, u_4, u_5) and 4 edges

The length of a path P is the sum of its edge weights along the path and is denoted by

$$w(u_1, u_k) = l(P) := \sum_{i=1}^{k-1} w(u_i, u_{i+1})$$

By $|P|$, we denote the number of edges along the path.

The distance between two nodes $u, v \in V$ is the minimal length of all paths P from u to v.

$$d_G(u, v) := \min\{ l(P) | P = \langle u, \dots, v \rangle \; path \; in \; G\} \; with \; \min \emptyset = \infty$$

If G is clearly indicated, it is omitted : $d_G(u, v) = d(u, v)$.

Note that it is possible that there might be more than one minimal paths from u to v. A minimal path P between two nodes u, v is called shortest path from u to v.

Let $V' \subseteq V$, then $G(V', E')$ with $E' = \{(x, y) \in E | x, y \in V'\}$ is a subgraph of G.

2.2 Data structures for graph representation

There are multiple data structures for graph representations and their use depends heavily on the characteristics of the input graph and the performance requirements of each specific application. In this thesis, we provide four representation options for the graph : the adjacency list representation, the static forward star representation, the dynamic forward star representation and the packed-memory graph representation. We assume the reader is more familiar with the first three. The fourth one is a new dynamic graph structure for large-scale transportation networks which provides unique features. It was developed by Zaroliagis, Mali, Michail, Paraskevopoulos in the University of Patras ([MMPZ12]).

2.2.1 Adjacency list representation

In graph theory and computer science, an adjacency list representation of a graph is a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbors of its vertex. The adjacency list representation of our graphs associates each vertex in the graph with a collection of its neighboring edges and all vertices are stored in a node list. It is implemented with linked lists of adjacent nodes. Figure 2.3 shows the adjacency list representation of the graph in Figure 2.2.
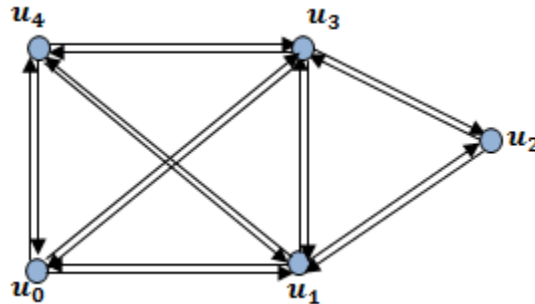
Figure 2.2 : A directed graph with 5 nodes and 16 edges, as given in paper [MMPZ12]
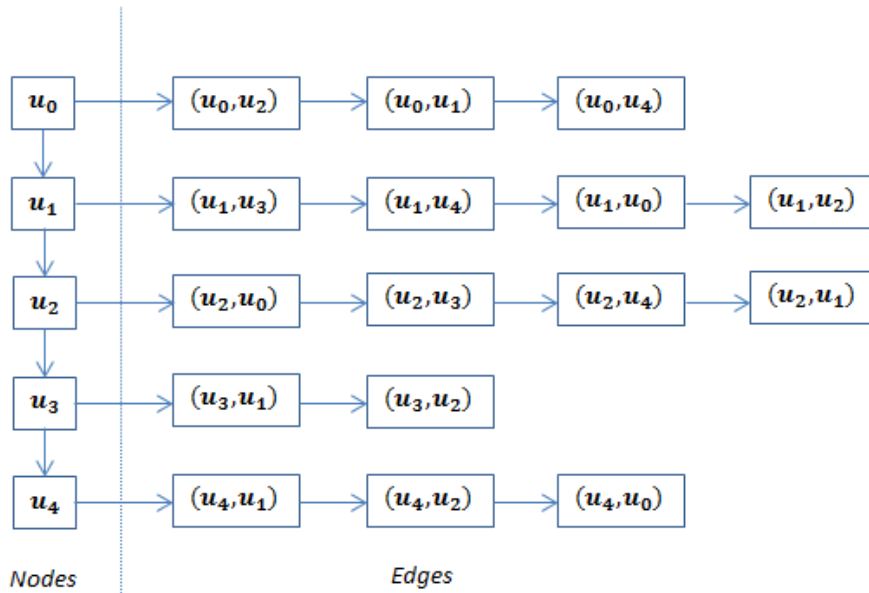
Figure 2.3 : Adjacency list representation of graph in figure 2.2, as given in [MMPZ12]

Adjacency list provides dynamicity in a way that it supports insertions and deletions of nodes and edges in $O(1)$ time complexity. However, it provides no guarantee on the actual layout of the graph in memory, since it is handled by the system's memory manager.

2.2.2 Forward star representation *(static and dynamic)*

A second representation we use for our graphs is the forward star which is a very interesting variant of the adjacency list and extensively used in several speed-up techniques. The forward star representation implements the node list of the adjacency list representation as an array and all adjacency lists are appended to a single edge array sorted by their source. The nodes and edges can be stored in consecutive, non-overlapping memory addresses which can then be scanned with maximum efficiency. We understand that the forward star is a very space-efficient data structure that allows fast traversal of the graph. Figure 2.4 depicts the forward star representation of figure 2.2.
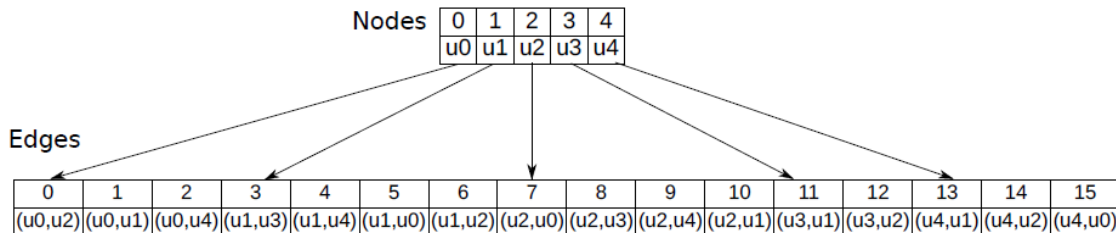
Figure 2.4 : The forward star representation of graph in figure 2.2 as given in [MMPZ12] paper.

The basic arrays have to be extended in order to incorporate necessary information for the nodes and the edges. To attach additional data to the nodes and edges, the entries of the node and edge vectors do not only contain pointers but structs. The same applies to the adjacency list representation.

The main disadvantage of forward star representation is that, in case of dynamic graphs, in order to insert an edge at a certain adjacency segment, all edges after the segment must be shifted to the right. This shift needs $O(m)$ time complexity. For this reason, a dynamic version of the forward star representation was developed. The adjacency segments has size equal to a power of 2, containing the edges and some empty cells at the end. So, when inserting an edge, if there are empty cells in the proper segment, the new edge is inserted. Otherwise, the whole segment is moved to the end of the edge array, and its size is doubled.

2.2.3 Packed-memory graph representation

The third structure we used for graph representation is the packed-memory graph (see also [MMPZ12]). As we said before, although the adjacency list provides insertions and deletions in $O(1)$, it does not provide any guarantee on the actual layout of the graph in memory. On the other hand, the forward star structure occupies consecutive memory addresses dedicated to the graph, which facilitates the scanning process of nodes and edges. Packed memory graph is a new structure that can combine the positive aspects both of the adjacency list and the forward star representation. The new structure is able to efficiently access consecutive nodes and edges,

to change and reconfigure its internal layout in order to improve the locality of the elements and to efficiently insert or delete nodes and edges (in cases of dynamic graphs). These three features reflect the compactness, agility and dynamicity of the data structure which are defined as follows :

- **Compactness :** The ability of scanning consecutive nodes and edges in an optimal way as far as time and memory transfers are concerned. The compactness of the packed-memory graph representation is comparable to the maximum efficiency of the forward star representation.
- **Agility :** The ability to reorder nodes and edges in allocated memory in order to increase the locality of reference. The various speed-up techniques implementations can give their desired node ordering as input to the packed-memory graph structure.
- **Dynamicity :** The ability to insert or delete edges and nodes in an optimal way in terms of time. The dynamicity of the new structure is comparable to the performance of the adjacency list representation when implemented as a linked list.

The packed-memory graph representation is based on a data structure, called packed-memory array. A packed-memory array maintains N ordered elements in an array of size $P = c \cdot N$, where $c > 1$ is a constant. Hence, the array contains N ordered elements and $(c - 1) \cdot N$ empty cells, called holes. The goal of a packed-memory array is to support efficiently insertions, deletions and scans by keeping the holes in the array uniformly distributed. This is accomplished by dividing the array in segments of size $\Theta(logP)$ such that a constant fraction of each segment contains holes. When a segment of the array becomes too full or too empty – depending on the density bounds imposed – its elements are spread out evenly within a larger interval by keeping their relative order. This process is called a rebalance of the (larger) interval (see also [MMPZ12]).
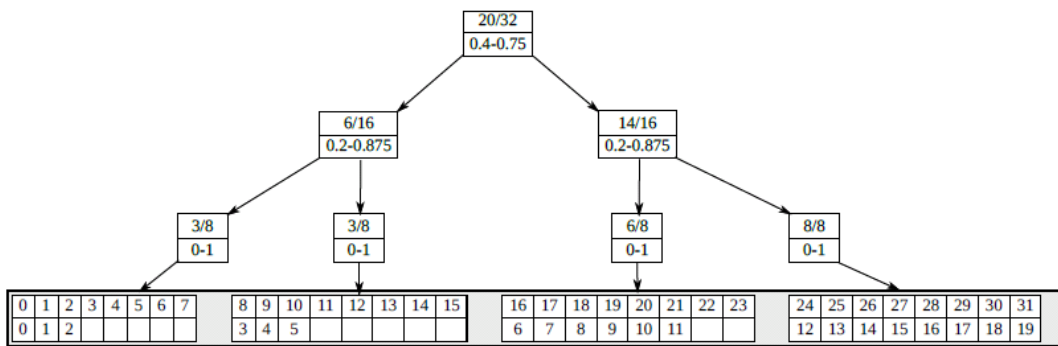
Figure 2.5 : Packed-Memory Array on the ordered set [0-19]. The array is divided in segments such that the number of them is a power of 2 and a perfect binary tree is built iteratively on top of these segments. This figure is taken from the paper [MMPZ12].

As mentioned before, the packed-memory graph representation is based on the packed-memory array data structure. The figure below illustrates the packed-memory graph representation.
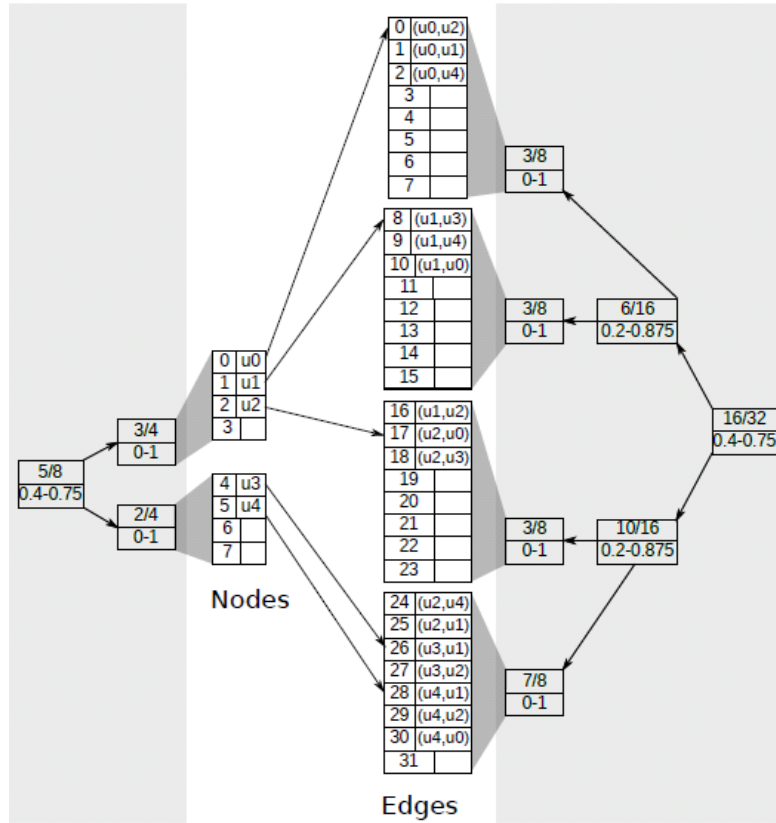
Figure 2.6 : Packed-Memory Graph representation as depicted in [MMPZ12]

This graph data structure stands as a good compromise between two extremes, the adjacency list representation which offers optimal dynamicity and the forward star representation which offers optimal compactness and agility. Especially, it is slower than the adjacency list in update time, but close to 30% faster in query times and a bit slower than the forward star in query time but over a million times faster in update time. Besides, one of the most important operations supported by the graph structure is the internal node reordering. That is to say, the structure can internally change the relative position of the nodes and the edges, a really important functionality since there are algorithms that have some information beforehand about the sequence of accesses of the elements in the graph and this can be exploited in speeding-up their performance (i.e. hierarchical speed-up techniques).Therefore, improving the locality of those important nodes in memory can give a performance speed-up in the execution.

2.2.4 Comparison of graph structures

In this section, the three graph structures are compared on the basis of the three performance features, namely compactness, agility and dynamicity.

	Adjacency List	Forward Star	Packed-memory Graph
Space	$O(m+n)$	$O(m+n)$	$O(c_m m + c_n n)$
Time			
Scanning S edges	$O(S)$	$O(S)$	$O(S)$
Inserting/Deleting an edge	$O(1)$	$O(m)$	$O(\log^2 m)$
Inserting a node	$O(1)$	$O(n)$	$O(\Delta \log^2 n)$
Deleting a node u	$O(\Delta)$	$O(\Delta m + n)$	$O(\Delta \log^2 m + \Delta \log^2 n)$
Memory Transfers			
Scanning S edges	$O(S)$	$O(1 + S/B)$	$O(1 + S/B)$
Inserting/Deleting an edge	$O(1)$	$O(1 + m/B)$	$O(1 + \frac{\log^2 m}{B})$
Inserting a node	$O(1)$	$O(1 + n/B)$	$O(1 + \frac{\Delta \log^2 n}{B})$
Deleting a node u	$O(\Delta)$	$O(1 + \frac{\Delta m + n}{B})$	$O(1 + \frac{\Delta \log^2 m + \Delta \log^2 n}{B})$

Table 1 : Comparison of space, running time and memory transfer complexity between the three graph data structures. B denotes the cache block and Δ denotes the maximum node degree. This table is taken from the paper [MMPZ12].

An adjacency list structure implemented with linked lists is a reasonable candidate for our graph representation, given that our graphs are static. It supports optimal insertions of nodes and the scanning of the edges is fast enough in practice. However, since there is no guarantee for the memory allocation scheme, the nodes and edges are most probably scattered in memory, resulting in many cache misses and less efficiency during scan operation, especially for the large-scale networks. Finally, it offers no support for any (re)-ordering of the nodes and edges [MMPZ12].

In contrast to the adjacency list, the forward star representation is optimal during the scan operations. Due to its layout, S consecutive edges are stored in at most $1 + \frac{S}{B}$ memory blocks. Hence, during a scan operation, the least amount of blocks is transferred into the cache memory. Moreover, its elements can be reordered in-line in a way that will favor the memory accesses of any algorithm. However, an insertion/deletion of a node or edge must shift all subsequent elements in the array in order to make space for the new element [MMPZ12].

A packed-memory graph representation is effective in all three features. The elements are stored as in the forward star representation, with only one difference: it keeps slightly larger arrays complemented with empty elements uniformly distributed within the array. Thus, it accomplishes efficient scanning of consecutive elements. Furthermore, it supports fast enough insertions and deletions of elements. Finally, it offers the element reordering in order to favor the memory accesses of each algorithm [MMPZ12].

The impact of these three representations on time complexity is going to be checked practically on multiple executions of the plateau and hybrid algorithm in large-scale road networks like the road network of Luxembourg, Belgium, Germany etc., in chapter 6.

2.3 Shortest path problem and route planning

In this section, we are going to present the shortest path problem in road networks and to introduce the basic concept of route planning in road networks.

2.3.1 Shortest path problem in road networks

A road network can easily be represented as a graph whether undirected, directed or mixed. The road junctions become the nodes of the graph and the road segments between nodes become the edges of the graph. Each edge is assigned a weight, e.g. the length of the road or an estimation of the time needed to travel along the road. Using directed edges it is also possible to model one-way streets. Such graphs are special in the sense that some edges are more important than others for long distance travel (i.e. highways). In graph theory, the computation of the shortest paths between two nodes is a classical problem. Actually, we can distinguish between several variants of this problem:

- point-to-point : compute the shortest-path length from a given source node $s \in V$ to a given target node $t \in V$
- single-source : for a given source node $s \in V$, compute the shortest-path lengths to all nodes $v \in V$
- many-to-many : for given node sets $S, T \subseteq V$, compute the shortest-path length for each node pair $(s, t) \in S \times T$
- all-pairs : a special case of the many-to-many variant with $S := T := V$

The most important algorithms for solving this problem are:

- Dijkstra's algorithm : solves the single-source shortest path problem
- Bellman–Ford algorithm : solves the single-source problem if edge weights may be negative
- A* search algorithm : solves for single pair shortest path using heuristics to try to speed up the search
- Floyd–Warshall algorithm : solves all pairs shortest path problem
- Johnson's algorithm : solves all pairs shortest path problem, and may be faster than Floyd–Warshall on sparse graphs

2.3.2 Route planning in road networks

While the above-mentioned algorithms compute optimal shortest paths with optimal theoretic time complexity, they are too slow to process real world data sets like large scale road networks, even on today's computers. However, only in recent years, computer hardware has become efficient enough to allow the handling of large networks like they occur in route planning. Consequently, during the past years, research focused on developing speed-up techniques to accelerate the basic shortest paths algorithms by reducing their search space. In this section we mention work on the subject of road networks.

Although, the first attempts to speed up Dijkstra's algorithm were conducted regarding timetable information on railway networks in 1999 (see [SWW99]), huge road networks were made publicly available in 2005 which led research toward road networks. This culminated in the 9th DIMACS Challenge on shortest paths [DGJ09] in 2006.

All modern speed-up techniques belong to one of 3 basic categories : Bi-directional search, goal-directed search and contraction.

In short, bi-directional search starts two Dijkstra searches, one from the source and one from the target. The latter is performed on the reverse graph and is labeled backward search in contrast to the forward search from the source. The final shortest path is then combined from partial paths obtained by the forward and backward searches. While this approach works well in time-independent networks where the edge weights are constant in the graph, adapting bi-directional routing to time-dependent networks where the edge weights are time-dependent functions is not straightforward .

As far as goal-directed search is concerned, the search is directed towards the target t by preferring edges that shorten the distance to t and by excluding edges that cannot possibly belong to a shortest path to t. There are different existing approaches. One is based on the A* algorithm by Nilsson and Raphael [HNR68], another on the enhanced A* by Goldberg et al. who introduce landmarks to compute feasible potential functions using the triangle inequality [GH05, GW05]. Their approach is called ALT and turns out as a very robust technique [BDW07]. The second goal-directed approach is using edge-labels to guide the search. Wagner et al. introduced in [WW03, WWZ05] a method called geometric containers where each edge contains a label that represents some geometric object containing all nodes to which a shortest path begins at the respective edge. During the query, edges that do not contain the target node can be pruned. This approach, refined by Lauther in [Lau04], is called Arc-Flags. Instead of geometric containers, the graph is partitioned into R regions and R edge-labels (arc-flags) are attached to every edge to indicate whether the respective edge is part of any shortest path leading into each region.

Regarding contraction there are a variety of approaches. Highway Hierarchies by Sanders and Schultes [SS05, SS06a] exploits the implicitly given hierarchy in road networks regarding different road categories. Contraction Hierarchies presented by Geisberger et al. [Gei08, GSSD08] is solely based on contracting the graph yielding a very efficient speed-up technique.

Furthermore, reach method reduces the search space of the graph. The reach of a node , $r(v)$, is a measure of centrality. In other words, a node with high reach value is more central to the graph in such a way that it is usually close to the middle of a long shortest path, whereas low-reach nodes are located rather near the end of shortest paths. If $r(v) < d(s, v)$ and $r(v) < d(v, t)$ holds, v cannot be on the shortest path from s to t, and does not need to be checked and settled by the query algorithm, so it can be pruned.

Generally, there are many combinations based on the afore-mentioned speed-up techniques. It turns out that the combinations of some techniques are more useful than of others. In particular, it has been observed that combining speed-up techniques from different categories is most promising whereas a combination of two similar techniques usually does not yield viable results since they tend to exploit the same aspects of the graph. For example, a combination of the two hierarchical techniques, i.e. Reach and Highway Hierarchies is not very promising.
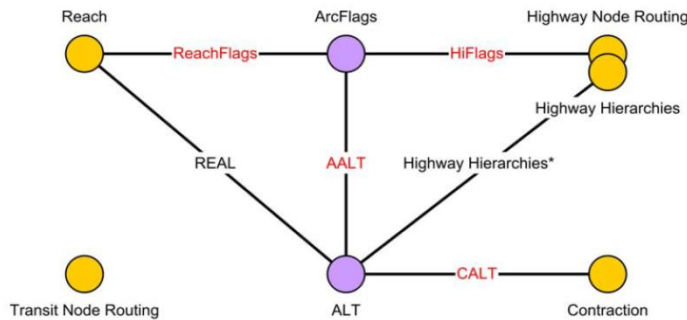
Figure 2.7 : Overview of the techniques and combinations as given in [Sch08]. Goal-directed are marked with purple and hierarchical are drawn in yellow. Edges denote existing combinations

2.4 Speed-up techniques

In the previous section, we introduced briefly the basic speed-up techniques to solve the shortest path problem in large road networks. In this section, they will be presented to a greater extent as a background to the general route planning problem. Note that, although the analysis of the speed-up techniques does not belong to the main object of this thesis, the development of the speed-up techniques is one of the most significant research fields in large-scale route planning and some of them can be easily incorporated as an extension to this thesis (which also deals with large-scale networks).

The development of speed-up techniques started when even the evolved computer hardware could not handle large-scale networks, like continental-sized networks, encountered in route planning, resulting to high query times. To encounter this problem, research in the past years focused on developing speed-up techniques for Dijkstra algorithm. The common goal of these techniques is to reduce the search space while still yielding optimal results. In other words, the majority of the speed-up techniques preprocess the input data, in order to accelerate the answer to single-source single-target shortest paths queries. This means that the solution method has two phases (in most times): a preprocessing phase, that computes useful information on the input graph and is applied only once, and a query phase, which computes the actual shortest paths using the output of the preprocessing phase to accelerate the search. For this reason, wherever feasible, the analysis is structured in a preprocessing phase and in a query phase.

It turns out that speed-up techniques are based on a few, basic concepts and thus can be categorized in bidirectional search, goal-directed search and contraction.

In this section, we present the basic speed-up techniques by introducing their algorithmic concepts in the context of uni-modal time-independent routing.

2.4.1 Bidirectional search

The bidirectional approach is probably the most evident idea to speed-up shortest path query from one source s to one target t. Ira Pohn was the first to design and implement it, but Andrew Goldberg et al. explained the correct termination conditions for the bidirectional version of Dijkstra's Algorithm.

 Bidirectional search executes two Dijkstra simultaneously, one forward from the source and the other backwards from the target. The forward search is performed in G, while the other in $\overleftarrow{G}$. The two Dijkstra computations (forward and backward direction) are interleaved as follows. Two priority queues are used to store the distances of the two Dijkstra. In each step, the Dijkstra which priority queue has the smallest key is executed. The algorithm terminates if a node becomes settled in one direction that has already been settled in the other direction. This node is called the meeting node. The shortest path can be derived from the information already gathered. The cost of the optimal route is computed as the $\min\{d_f(s,v) + d_b(t,v)\}$, for all v visited in both directions(the meeting node does not necessarily lie on the shortest path).

Regarding the illustration of the search space in *Figure 2.8*, it can be perceived why a bidirectional search is usually faster than a normal Dijkstra query. In road networks, where search spaces will take a roughly circular shape, we can expect a speedup around two – one disk $d(s,t)$ has twice the area of two disks with half the radius.

(a) Uni-directional Dijkstra   (b) Bidirectional Dijkstra
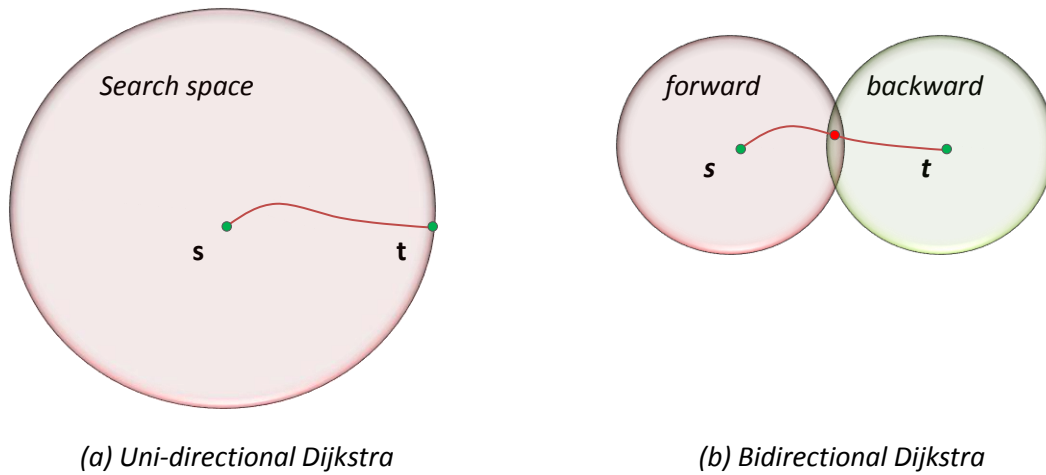
Figure 2.8 : Comparison of uni-directional with bidirectional search
The respective search spaces are illustrated by disks around source and target node.

Bidirectional Dijkstra is one of the most known and useful speed-up techniques and it is widely combined with many other techniques either based on goal-directed search or contraction.

2.4.2 Goal-directed search

Goal-directed approaches direct the search towards the target t by preferring edges or nodes that shorten the distance to t and by excluding edges or nodes that cannot possibly belong to a shortest path to t—such decisions are usually made by relying on preprocessed data.

2.4.2.1 A* search

In computer science, A* is a computer algorithm originated from artificial intelligence and widely used in pathfinding and graph traversal. It was first described in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael [HNR68].

The A* algorithm is a modified Dijkstra algorithm that applies additional information to improve the performance. A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). Let $G = (V, E)$ be an arbitrary graph. As A* traverses G, it follows a path of the lowest known heuristic cost, keeping a sorted priority queue of alternate path segments along the way.

Specifically, let $\pi : V \to \mathbb{R}$ be an arbitrary potential function on the graph nodes to estimate distances between them. Then, a reduced weight function $w_\pi(u, v)$ of a path $P\langle u, ..., v \rangle$ can be defined as the sum of two parts

- the path-cost function $l(P) = w(u, v)$, which is the cost from the starting node u to the current node v
- an admissible "heuristic estimate", $\pi(v) - \pi(u)$, of the distance from the start node to the goal

Consequently, $w_\pi(u, v) = w(u, v) + \pi(v) - \pi(u)$.

Note that the length of an arbitrary $u - v$ path is only changed by a constant value $\pi(v) - \pi(u)$ when $w_\pi(\cdot)$ is applied. This potential function $\pi(\cdot)$ is called feasible if the reduced edge weights $w_\pi(\cdot)$ are non-negative for all edges $e \in E$. The potential $\pi(u)$ is a lower bound on the distance $d(u, t)$, if $\pi(t) \leq 0$ holds and $w_\pi(\cdot)$ is feasible .

The A* search applies a feasible potential function $\pi(\cdot)$ to speed-up $s - t$ queries. The basic structure of Dijkstra's algorithm is retained but priority keys are changed to $key(u) = d(s, u) + \pi(u)$. Thus, in each step the node u is settled with the shortest estimated path from the source to the target via u. Figuratively speaking, nodes that potentially lead closer to the target are preferred, whereas the other nodes are ignored.

This approach is equivalent to performing a classical Dijkstra on a graph with weights derived from the weight function $w_\pi(u, v)$. Since the length of arbitrary $u - v$ paths is only changed by a constant value $\pi(v) - \pi(u)$, the priority keys become $key(u) = d(s, u) + \pi(u) - \pi(s)$. This yields the same sorting as in priority queue above, since $\pi(s)$ is constant. Therefore, running a shortest-path search on the normal graph is equivalent to running one on the graph with reduced edge weights.

If a graph layout is given, the Euclidian distance to the target node is a suitable choice for a feasible potential function, as illustrated in the figure 2.9. This works only as long as the metric in the graph is also geographical distance since then geographical distance is guaranteed to be a lower bound of any path from v to t.
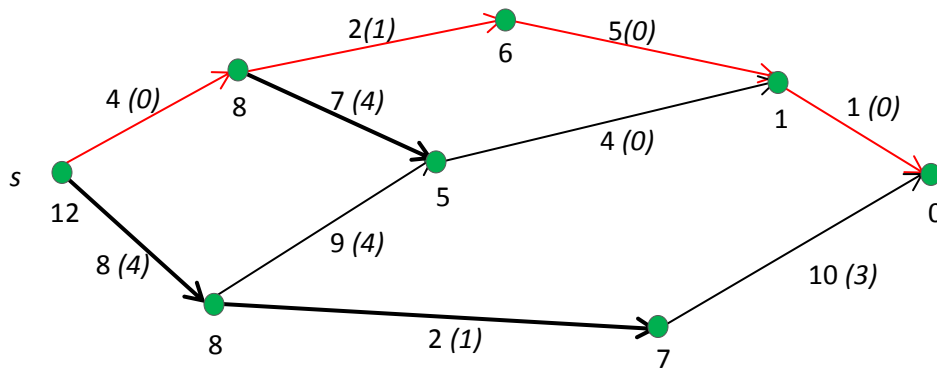
Figure 2.9 : Graph with a feasible potential function, inspired by [Sch08]. Potential values are written below the nodes and reduced weights are given in brackets next to the normal edge weights. The shortest path from s to t is shown in red and the associated search space is drawn in bold.

The two potentials are called consistent if the reduced weight function is the same for both. This is true if $\pi_f + \pi_b = const$ holds. If they are not consistent, the search cannot be stopped when the two search spaces meet since both directions use different weight functions and as a result new approaches should be implemented for the bidirectional A* search.

2.4.2.2 ALT algorithm

The ALT algorithm, which has been introduced by Goldberg and Harrelson in 2004, is a variant of A* and stands for A* search, Landmarks and Triangle inequality.

The ALT algorithm is an A* search with a far simpler potential function. Its main idea is to use landmarks and triangle inequality to produce feasible lower bounds. Furthermore, it is not dependent on the availability of additional layout information.

Preprocessing . In the preprocessing phase, a small set of nodes L is chosen and labeled as landmarks. Moreover, an exact distance table with the distances to/from every landmark $l \in L$, for all nodes v, is computed. The number of landmarks is usually set between 16 and 64. Because distances on G form a metric, the following instances of the triangle inequality hold :

$$d(u,t) + d(t,l) \geq d(u,l) \quad \text{and}$$

$$d(l,u) + d(u,t) \geq d(l,t)$$
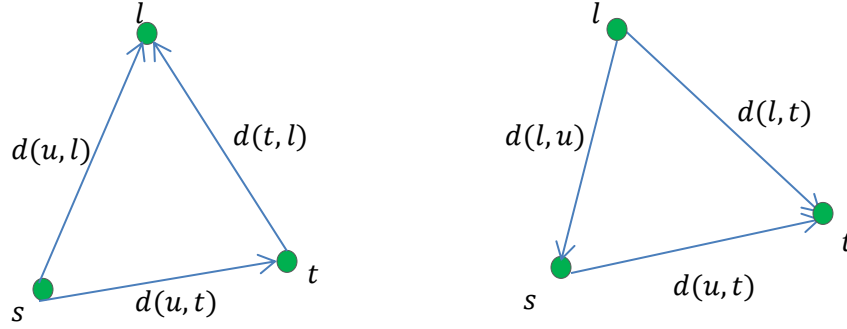
Figure 2.10 : The two pictures show the application of the triangle inequality

Query . During the query, a lower bound is computed according to $d(u,t) \geq d(u,l) - d(t,l)$ (implying distance to a landmark l) or $d(u,t) \geq d(l,t) - d(l,u)$ (implying distance from a landmark l). Specifically,

$$\pi_l(u) := \max\{d(u,l) - d(t,l), d(l,t) - d(l,u)\} \leq d(u,t)$$

The best lower bound π can be obtained by using the landmark yielding the greatest lower bound according to

$$\pi(u) := max_{l \in L} \max\{d(u,l) - d(t,l), d(l,t) - d(l,u)\}$$

With $d(u,l)$ and $d(l,u)$ precomputed for each landmark $l \in L$ and every node $u \in V$, the reduced cost graph G_π is then computed implicitly by altering the key of u in the priority queue to $d(s,u) + \pi(u)$.

Consequently, the only difference to Dijkstra algorithm is that instead of using $d(s,u)$ as keys in the priority queue, we use the cost reduced distance function $d(s,u) + \pi(u)$. However, previous experiments revealed that computing the lower bound with respect to all landmarks produces too much overhead during the query. For that reason, we apply triangle inequality only on a subset $L_{active} \subseteq L$. We usually restrict the cardinality of L_{active} to 2. The landmarks that are set active depend on the query and are determined in the beginning using $\pi_l(s)$. Furthermore, every k iterations of the algorithm we update the set of active landmarks by rechecking which landmarks yield the best lower bound for the currently settled nodes.

2.4.2.3 Arc flags

Besides the afore-mentioned goal-directed algorithms, the Arc-Flag approach (or edge labels) is another speed-up technique for Dijkstra algorithm. Although it belongs to goad-directed approaches, the underlying idea is different. It was first introduced by Lauther [Lau97, Lau04] and has ever since been studied and revised several times.

The basic intuition of the algorithm will be given with an example. If a driver is in Munich and wants to go to Berlin, he will follow mostly roads, their direction is to the north. Based on this idea, Arc-Flags try to find paths that can be pruned during the search, thus, yielding a smaller search space. This approach aims to put appropriate flags on the edges so as to use only the relevant edges during an s-t query, these leading to the direction of the goal/target.

Preprocessing . Arc-Flags require two-phase preprocessing**.**

First, the graph $G(V,E)$ is partitioned in a fixed number of cells or regions $R = (R_1, R_2, ..., R_k)$. An example of a graph partitioned into four regions is shown in *Figure 2.11.* In a s-t query, we shall refer to the region or the cell that the target belongs to as target region or target cell T. Thus, every node belongs to only one region.
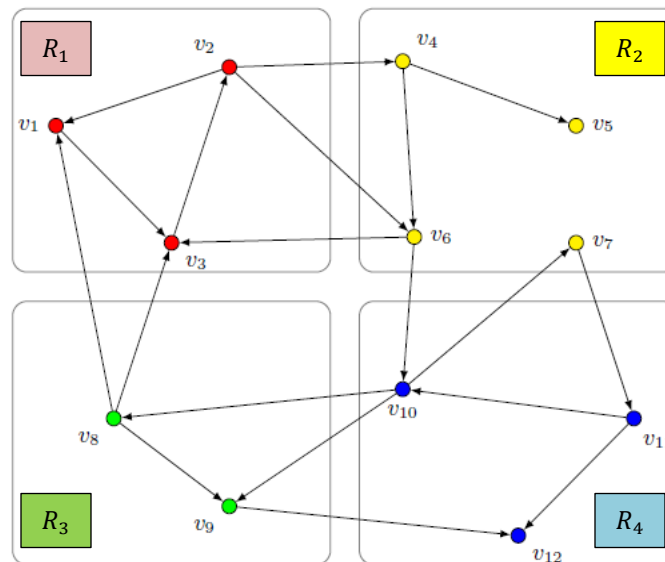
Figure 2.11 : A simple example of a partition in a graph G taken from [Bau11] . G is divided into four regions $R_1 = \langle u_1, u_2, u_3 \rangle$, $R_2 = \langle u_4, u_5, u_6, u_7 \rangle$, $R_3 = \langle u_8, u_9 \rangle$, $R_4 = \langle u_{10}, u_{11}, u_{12} \rangle$

Given the graph G and the partition R, we assign a distinct number in $\{1, ..., k\}$ to each region and define a mapping $r : V \rightarrow \{1, ..., k\}$ such that $r(u)$ is the number of the region that u belongs. Storing this information clearly requires space linear in n.

Regions should be divided properly so as to be nonoverlapping and compact. So, as far as the type of partition is concerned, experiments show (see [HKMS06, MSS+06]) that it has a major impact on query performance of the Arc-Flags query algorithm. While in fact any partition works, a 'good' partition should have the following properties. First, the cells of the partition should be connected. This helps the goal-direction of Arc-Flags. Moreover, the number of boundary nodes (i.e., nodes that are incident to edges connecting different cells) should be low. The main reason is, as we see soon, that a high number of boundary nodes imply a high

preprocessing effort. It should be noted that the geometric partitioning methods in [HKMS06, MSS+06] fulfill the first claim, while the second is not considered. These do not require geographical information attached to the nodes and compute the partition solely based on the structure of the graph. We omit further technical details and only like to point out that from the tested partitioning methods, METIS [Kar07], PARTY [MS04] and SCOTCH [Pel07], the latter yields the most promising results for Arc-Flags.

The second phase of the preprocessing is the computation of the edges flags. Thus, we enrich each edge e of the graph by a vector $F_e(\cdot)$ of k binary flags construed as boolean values. Each of the k boolean values corresponds to the relevant region and indicates if that edge is part of a shortest path to any node of that region. For example, if $F_{(u_1,u_2)}$ is equal to (1,0,0,0), edge (u_1,u_2) belongs to every shortest path with target node lying on region 1, but it does not belong to any shortest path to regions 2, 3 and 4. Thus the additional memory consumption necessary for these arc-flags amounts to $k \cdot m$ bits and therefore is linear to m. We interpret that $F(u,v)(r(t)) = 1$ means that the edge (u,v) might be important in any query with the target node t (belonging in every region R_i). Let $SP_{s,t}$ denote the set of all shortest s-t paths in a given graph. Every flag is then supposed to fulfill the following property to retain correctness of the query algorithm.

$$\forall\, s,t\, \in V : SP_{s,t} \neq \emptyset \implies \exists P \in SP_{s,t} : \forall (u,v) \in P : F_{u,v}\big(r(t)\big) = 1$$

By satisfying this expression we ensure that for any pair of source and target nodes, there exists at least one shortest path for which the flags corresponding to the target region are set to 1 on all of its edges.

The simplest and most naive way to compute arc-flags with respect to region R_i is the following. First, we initialize the ith flag on all edges except those edges with their tail inside R_i with false. Now we consecutively grow a full backward shortest path tree from every node $v \in R_i$, setting the ith flag to true for every edge that is contained in the tree. Arc-flags once set to true are never changed back to false. To complete this approach for every region, we end up computing $|V|$ full backward shortest path trees in $\overleftarrow{G}$ which is too slow for large graphs (see also [Lau04]).

A faster method only uses boundary nodes. The key observation is that a shortest path with target region R_i has to enter the region at some point. Hence, it is sufficient to compute backward shortest path trees from the boundary nodes (instead of the nodes in the region). This decreases the preprocessing time significantly [Lau04]. However, preprocessing time is still rather high (several hours, even up to days on large graphs), which is the major drawback of the Arc-Flags approach. Thus in [HKMS06] another approach is presented using centralized shortest path trees which decreases preprocessing time significantly.

Figure 2.12 : Arc-flags after preprocessing, as given in [Bau11]

Query . A unidirectional Arc-Flags query is a modified Dijkstra operating on the input graph. For a random s-t query, it first determines the target cell T, and then relaxes only those edges with set flag for cell T. Note that compared to plain Dijkstra, an Arc-Flags query performs only one additional check.

2.4.3 Contraction

The underlying idea of contraction is based on the hierarchy of roads. In other words, in a s-t query, when the driver is close to the source and the target, he should take into account mainly residential roads, when he is a bit further, mainly avenues and when he is even further away, mainly motorways.

So, contraction is a method that leads to a reduction concerning the size of the graph, either, regarding nodes (node reduction) or edges (edge reduction). A graph $G = (V, E)$, as generally used in this field of research, contains a lot of nodes that have very few connections to the other nodes. The goal of the contraction is to identify these nodes and to remove them, but retaining shortest-path distances between the remaining nodes by inserting additional shortcuts.

2.4.3.1 Highway hierarchies

Highway Hierarchies introduced in [SS05] and refined in [SS06a] is one of the first speed-up techniques using contraction based methods.

Preprocessing . Highway Hierarchies group nodes and edges in a hierarchy of levels by alternating between two procedures: Contraction (i.e., node reduction) removes low degree nodes by bypassing them with newly introduced shortcut edges (see also [SS05]). In particular, all nodes of degree one and two are removed by this process. Edge reduction removes non-highway edges, i.e., edges that only appear on shortest paths close to source or target. More specifically, every node v has a neighborhood radius $r(v)$, we are free to choose. An edge (u, v) is a highway edge if it belongs to some shortest path P from a node s to a node t such that (u, v) is neither fully contained in the neighborhood of s nor in the neighborhood of t, i.e., $d(s, v) > r(s)$ and $d(u, t) > r(t)$. In all our experiments, neighborhood radii are chosen such that each neighborhood contains a certain number H of nodes. H is a tuning parameter that can be used to control the rate at which the network shrinks.

Query . The query algorithm is very similar to bidirectional Dijkstra search with the difference that certain edges need not be expanded when the search is sufficiently far from source or target.

HHs were the first speedup technique that could handle the largest available road networks giving query times measured in milliseconds. There are two main reasons for this success: under the above contraction routines, the road network shrinks in a geometric fashion from level to level and remains sparse and near planar, i.e., levels of the HH are in some sense self-similar. The other key property is that preprocessing can be done using limited local searches starting from each node. Preprocessing is also the most nontrivial aspect of HHs. In particular, long edges (e.g. long-distance ferry connections) make simple minded approaches far too slow. Instead we use fast heuristics that compute a superset of the set of highway edges.

Routing with HHs is similar to the heuristics used in commercial systems. The crucial difference is that HHs are guaranteed to find the optimal path. This qualitative improvement actually makes HHs much faster than the heuristics. The latter have to make a precarious compromise between quality and size of the search space that relies on manual classification of the edges into levels of the hierarchy. In contrast, after setting a few quite robust tuning parameters, HH preprocessing automatically computes a hierarchy aggressively tuned for high performance.

2.4.3.2 Contraction hierarchies

Contraction Hierarchies [Gei08, GSSD08] is a further development based on Highway Hierarchies, yielding a high-performance speed-up technique solely based on the concept of contraction.

Preprocessing .

Node Reduction . The node-reduction when applied once, divides the graph into two parts: the

core and the component. In the beginning all nodes are considered to belong to the core. Iteratively, nodes are bypassed according to a certain order, they get extracted from the core and eventually belong to the component of the graph, until no further nodes are bypassable (the criterion to which nodes are selected for bypassing is not discussed at this point). A node n is bypassed by removing it from the graph along with all of its ingoing edges $I(n)$ and outgoing edges $O(n)$. For each pair of removed edges $(u,n) \in I(n)$ and $(n,v) \in O(n)$ with $u \neq v$, a shortcut (u,v) is inserted into the graph. Its weight is set to the sum of the weights of the removed edges: $w(u,v) = w(u,n) + w(n,v)$. If there already was an edge e in the graph, connecting u and v, the shortcut is not inserted. But if the weight of the shortcut would have been smaller than $w(e)$, it is used instead. Finally, the node n and all its incident edges are deleted from the graph. Note that the node-reduction routine preserves correct distances between two arbitrary core nodes. The obtained core graph is denoted by $G_{core} = (V_{core}, E_{core})$, while the component is defined by $G_{comp} = (V_{comp}, E_{comp})$ where $V_{comp} := V \setminus V_{core}$ and $E_{comp} = E \setminus E_{core}$.

(a) before   *(b) after*   *(c)edge-reduction (v,w)*
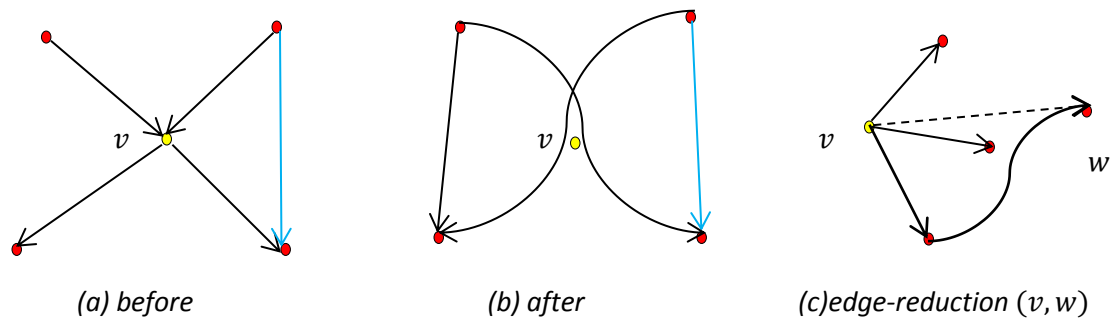
Figure 2.13 : Figures (a) and (b) illustrate the bypass operation during node reduction, inspired by [Paj09] thesis. For each pair of incoming and outgoing edges at v, a shortcut is inserted. If the shortcut e is already contained in the graph (blue edge), the weight on the shortcut is set to the minimum weight of e. Figure (c) illustrates edge-reduction. The bold path from u to w is shorter than the edge $e = (v,w)$(dashed line), thus, e can be deleted.

Edge Reduction .
Note that the node-reduction routine potentially adds shortcuts not needed for keeping the distances in the core correct. Hence, we perform an edge-reduction directly after node-reduction, similar to [SWW99]. We grow a shortest path tree from each node u of the core. We stop the growth as soon as all neighbors t of u have been settled. Then we check for all neighbors t whether u is the predecessor of t in the grown partial shortest path tree. If u is not the predecessor, we can remove (u,t) from the graph because the shortest path from u to t does not include (u,t). In order to remove as many edges as possible we favor paths with more hops over those with few hops. In order to limit the running time of this procedure, we restrict the number of priority-queue removals to 10.000. Hence, we may leave some unneeded edges in the graph.

Query . While the query algorithms of the previous techniques turned out easy, the contraction routine requires a more complicated query algorithm (see also [Paj09]). As input we are given a graph $G = (V, E)$ with designated core-nodes. For an $s - t$ query the algorithm works in two phases. The first phase operates on the component part of the graph, while the second operates only on the core. Phase one instantiates a bi-directional search on the component of G. This is achieved by not relaxing edges that are contained in the core (i.e., edges $e = (u, v)$ for which both u and v have the core flags set). Note that by these means we in fact settle core nodes, we just abort the search as soon as they are first hit (if either s or t are core nodes, the forward resp. backward search terminates immediately). The set S of core nodes that are hit by the forward search is called set of core-entry-nodes, while the set T of core nodes hit by the backward search is called the set of core-exit-nodes. Phase one terminates if one of the following conditions holds.

- Both, the forward and backward priority queues are empty.
- There has been a $s - t$-path P found for which it holds that

$$l(P) \leq d\left(s, v_{entry}^{min}\right) + d(v_{exit}^{min}, t)$$

where $v_{entry}^{min} \in S$ denotes the core-entry-node with minimal distance from s in G, while $v_{exit}^{min} \in T$ denotes the core-exit-node with minimal distance from t in $\overleftarrow{G}$.

If phase one is aborted due to the second condition, we can stop the query and output the computed path P as shortest s-t-path. In this case, the shortest path solely uses nodes of the component. So for the rest of this paragraph we assume that no path P in the component has been found by phase one. In this case, phase two of the algorithm is instantiated with a many-to-many $s - t$ query only relaxing edges contained in the core. However, the forward (and backward) queues are re-filled such that the initial keys are set to the distances (from s respectively t) computed in phase one. The algorithm used in phase two is not specified. Thus, contraction yields a modular design that allows combination with an arbitrary speed-up technique applied on the core. The final $s - t$-path is then combined by determining the minimal $s - T - t$ path $P_{s,v,t}$ for every node $v \in T$ where the length is computed by

$$l(P_{s,v,t}) := d(s, v) + d(v, t)$$

(a) Phase I   (b) Phase II
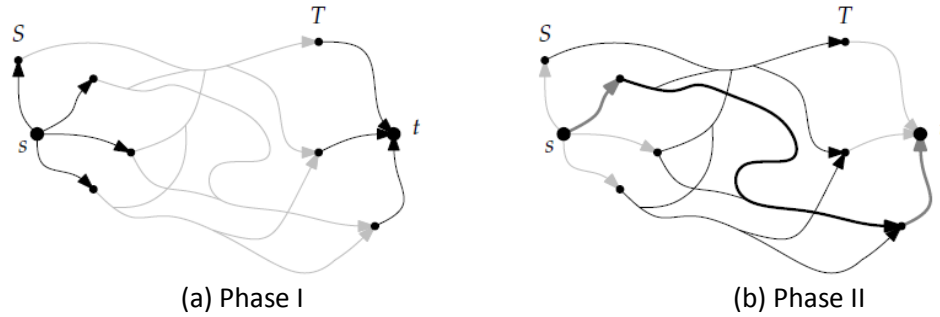
Figure 2.14 : Illustration of the core-based routing algorithm as given in [Paj09]. Phase I conducts a bidirectional Dijkstra until all core-entry resp. core-exit nodes have been reached. In phase II, an s-t query is performed on the core graph. The shortest path is then combined by taking from all s-T-t paths the one with the minimum length.

2.4.3.3 Reach

The notion of reach in the context of graphs was first introduced by Gutman in [Gut04]. It can be applied either to nodes or to edges, with the latter being more effective but also needing more space. Here, the Reach algorithm is described using node-reaches, but most of the explanation can be easily adapted for edge-reaches.

Definition 1 . The reach of a node v, denoted by $r(v)$, is defined as the maximum, over all shortest u–w paths containing v, of $min\{d(u,v), d(v,w)\}$.
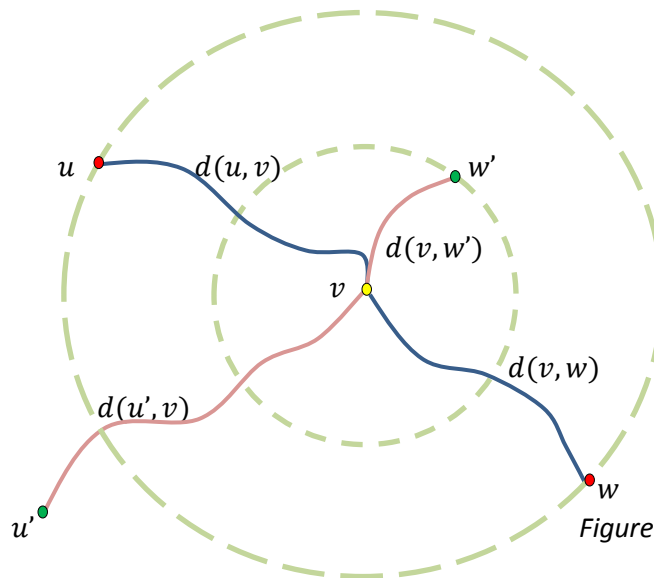
Figure 2.15 : Illustration of reach

So, $r(v) := \max_{P_v} \min\{d(u,v), d(v,w)\}, u, w \in V$

30

Therefore, the reach value is a measure for the centrality of the node. In other words, a node with a high reach value is more central to the graph in such a way that it is usually close to the middle of a long shortest path, whereas low-reach nodes are located rather near the end of shortest paths. On a road network, this differentiation corresponds to important highway roads and non-relevant local roads.

Preprocessing . The reach values for each node are computed in a preprocessing step and can be used during the query to prune the search space. If $r(u) < d(s,u)$ and $r(u) < d(u,t)$ holds, v cannot be on a shortest path from s to t and does not need to be touched or settled by the query algorithm. Note that upper bounds on reaches $r(u)$ and lower bounds on distances $d(u,v)$ suffice for the condition to hold.

Query . The query algorithm implements Dijkstra's algorithm with pruning based on reaches. Whereas reach values are precomputed and available during the query, and distances from the source to a node v are automatically given by the query, the distance from v to the target is more difficult to be obtained. Gutman [Gut04] suggested using Euclidean distances to compute lower bounds as is done by the A* search. A graph layout is required to compute the Euclidean distances and if the weight function is not based on a distance metric, this approach usually does not produce good lower bounds. Goldberg et al. found a more promising way to obtain lower bounds implicitly by using a bidirectional query together with reach-based pruning (see also [Fuc10]).

2.4.4 Summary

In this section 2.4, we presented speed-up techniques for Dijkstra's algorithm which are an integral ingredient in large-scale route planning so as to minimize the query times. These techniques can also apply to the alternative route problem either separately or combined.

In general, each technique (or speed-up category) has its advantages and its disadvantages which can be taken into account depending on the application, the specifications for memory consumption and execution times and generally the problem under consideration [Del09]. In particular, ALT is easily adapted to dynamic scenarios, is robust to the input and its preprocessing algorithm is easy enough. However, it yields high memory consumption. The advantage of Arc-Flags is the exceptional query performance. Preprocessing phase is based on Dijkstra search, while the query algorithm performs only one additional check compared to plain Dijkstra. However, the time consuming preprocessing phase (needs more than 17 hours to preprocess a continental-sized road network) is a crucial drawback. Hierarchical speed-up techniques are quite successful due to the iterative contraction of the input. It turns out that, at least in road networks, it is often sufficient to perform extensive preprocessing only on a small subgraph of the input, the core. Thus, combining techniques belonging to different categories, we can benefit from the advantages of each one, achieving great results.

CHAPTER 3

ALTERNATIVE ROUTE ALGORITHMS AND ALTERNATIVE GRAPHS

As we mentioned in the beginning, today's requirements for routing services, be it in-car or as a web-service, ask for more than just computing the shortest path. Thus, it is desirable not only to compute a single path to a user, but instead a set of paths. This chapter introduces us to the main subject of our study, that of finding alternative routes in graphs. Given an initial graph G and a pair of source and target nodes, the aim is to compute a subgraph of G resulting from the union of the alternative routes found from s to t. So, initially, we define this resulting subgraph, called Alternative Graph (AG) and then we continue by discussing basic methods used for finding alternative routes and as a result creating AGs. K-shortest paths, disjoint paths, pareto-optimal paths, penalty method are some approaches in this direction. The chapter continues by stating where these algorithms are used and which are their main drawbacks that led the study in defining certain criteria for the alternative routes and developing new approaches, presented in chapter 4.

3.1 Alternative graph *(AG)*

No matter which approach is used for the computation of the alternative routes, the final goal is the creation of the alternative route subgraph, denoted by AG, which consists of the paths computed by the approach.

The initial graph is the representation of a network and the desirable output is a subgraph resulting from the union of several alternative paths from source to target. In general, these alternatives can share nodes and edges and subpaths of them can be combined to new alternative routes. So, more formally, let $G(V, E)$ be the input graph with edge weight function $w : E \rightarrow R_+$. For a pair of source and target nodes, s, t, an alternative route graph or more simply, an alternative graph, $AG = (V', E')$ is a graph consisting of paths starting from s and ending to t that represent alternative roads in the network. $V' \subseteq V$ such that for every edge $e \in E'$, there exists a simple $s - t$ path in AG containing e and none of the nodes is isolated.

3.2 K-shortest paths

K-shortest paths is a widely used approach to find alternative paths. As the name states, the k-shortest-paths problem, for a given k and a given source-destination pair in a graph is to list k

paths in the graph with minimum total length. There are two types of k-shortest-paths network problems. The first is to find k paths from the origin to the sink that have the shortest lengths, in which loops are allowed. Hoffman and Pavley ([HP59]), Bellman and Kalaba ([BK60]), Sakarovitch ([Sak66]) proposed different algorithms for solving this type of problem, but Eppstein's algorithm([Epp94]) achieves the best running time complexity. The second type of problem is to find k paths from the origin to the destination that have the shortest lengths, in which no loops are allowed. The available algorithms for solving this type of problem are proposed by Bock, Kantner and Haynes ([BKH57]), Pollack ([Pol61]), Clarke, Krikonian and Rausan ([CKR63]), Sakarovitch ([Sak66]) and others. However, the best running time for this case is attributed to Yen([Yen71]).

3.2.1 K-shortest paths with loops *(by Eppstein)*

Eppstein solved efficiently the problem if k-shortest path problem in 1994. Paths are allowed to have loops.

Preliminaries . Let $G(V,E)$ be a graph with $n = |V|$ the number of vertices and $m = |E|$ the number of edges. Self-loops and multiple edges in the graph are allowed, so m maybe larger than $\binom{n}{2}$. Furthermore, for the purposes of the algorithm, a heap is a binary tree in which vertices have weights, satisfying the restriction that the weight of any vertex is less than or equal to the minimum weight of its children. More generally, a D-heap is a degree-D tree with the same property. Thus the usual heaps are 2 –heaps.

The algorithm described below, does not output each path it finds explicitly as a sequence of edges, but implicitly. The representation is similar in spirit to those used for the k minimum weight spanning trees problem: for that problem, each successive tree differs from a previously listed tree by a swap, the insertion of one edge and removal of another. The implicit representation consists of a pointer to the previous tree, and a description of the swap. For the shortest path problem, each successive path will turn out to differ from a previously listed path by the inclusion of a single edge not part of a shortest path tree, and appropriate adjustments in the portion of the path that involves shortest path tree edges. Our implicit representation consists of a pointer to the previous path, and a description of the newly added edge.

Eppstein's algorithm goal is to create appropriate data structures (in particular a path graph $P(G)$ and a heap $H(G)$) from which we can reconstruct k-shortest paths from s to t. Thus, it is necessary to explain some basic concepts, to see how a path is represented by a heap, to define the path graph $P(G)$ and the path heap $H(G)$ and finally find the k shortest $s - t$ paths in G.

Steps of the algorithm .
At first, the algorithm constructs the destination tree T. In other words, it finds the shortest path from each vertex to the destination node, running a Dijkstra on the backward graph $\overleftarrow{G}$.

Basic Concepts . Given an edge e in G, $\delta(e) = l(e) + d(head(e), t) - d(tail(e), t)$ is defined. Intuitively, $\delta(e)$ measures how much distance is lost by diverging from the shortest path tree or alternatively by being «sidetracked» along edge e instead of taking a shortest path to t. For any $e \in G$, $\delta(e) \geq 0$. For any $e \in T$, $(e) = 0$.

For any path P in G, some edges of P may belong to T, and some others may belong to $G - T$. Any path P from s to t is uniquely determined solely by the subsequence $sidetracks(P)$ of its edges in $G - T$ (sidetracks are edges that diverge from the shortest path tree). For a given pair of edges in $sidetracks(P)$, there is a uniquely determined way of inserting edges from T so that the head of the first edge is connected to the tail of the second edge. A sequence of edges in $G - T$ may not correspond to any $s - t$ path, if it includes a pair of edges that cannot be connected by a path in T. If $S = sidetracks(P)$, we define $path(S)$ to be the path P. The implicit representation will involve these sequences of edges in $G - T$. For any nonempty sequence S of edges in $G - T$, let $prefix(S)$ be the sequence formed by the removal of the last edge in S. If $S = sidetracks(P)$, then $prefix(S)$ will define a path $prefpath(P) = path(prefix(S))$. We next show how to recover $l(P)$ from information in $sidetracks(P)$.

For any path p from s to t ([Epp94]),

$$l(P) = d(s,t) + \sum_{e \in sidetracks(P)} \delta(e) = d(s,t) + \sum_{e \in P} \delta(e)$$

Representation of Paths by Heap . The representation of $s - t$ paths discussed in the previous section gives a natural tree of paths. Every path P is represented by a path in the tree in which the parent is $prefpath(P)$ and the child is $lastsedge(P)$. The degree of any node in this path tree is at most m, since there can be at most one child, corresponding to each possible value of $lastedge(P)$. The possible values of $lastedge(Q)$ for paths Q that are children of P are those edges in $G - T$ that their tails rely on the path starting from node $head(lastedge(P))$ and ending to destination node t, along the shortest path tree T. Note that if G contains cycles, the path tree is infinite. The path tree is heap-ordered. However, since its degree is not constant, we cannot find the k smallest weight vertices (as it is usually done in any heap, in time $O(k)$). Instead, we form a heap by replacing each vertex v of the path P with an equivalent bounded-degree subtree (heap of edges at vertices).

Heaps of Edges at Vertices . For each vertex v, we form a heap $H_G(v)$ containing all edges with tails on the path form v to t, ordered by $\delta(\varepsilon)$. This heap H_G is used to modify the path tree that was described before. Specifically, every node v of the path tree is replaced by a copy of $H_G(head(lastedge(P)))$. The procedure is described in detail below.

$Out(v)$ contains all edges leaving a vertex v in a graph G which are not part of a shortest path in G. They are heap-ordered by $\delta(e)$ that represents the distance lost when using this edge e instead of the one on a shortest path (shortest paths have been found at the computation of tree T) . Now, we build a 2-min-heap $H_{out}(v)$ by heapifying the set of edges $out(v)$ according to their $\delta(\varepsilon)$ for any node $v \in V$.

Then, we build for each vertex v, a balanced heap $H_T(v)$, containing only the roots $outroot(w)$ of the heaps $H_{out}(w)$, for each vertex w on the path from v to t. $H_T(v)$ is formed by inserting $outroot(v)$ into $H_T(next_T (v))$. Since insertion into a balanced heap can be performed with $O(log\ n)$ changes of pointers on a path from the root of the heap, we can store $H_T(v)$ without changing $H_T(nextT (v))$, by using an additional $O(log\ n)$ words of memory to store only the nodes on that path.

Now, we can build $H_G(v)$, by making each node $outroot(w)$ in $H_T(v)$ point to an additional subtree, namely to the rest of heap $H_{out}(w)$. $H_G(v)$ can be constructed at the same time as $H_T(v)$. $H_G(v)$ is thus a 3-heap as each node includes at most either two edges from $H_T(v)$ and one edge from $H_{out}(w)$, or no edges from $H_T(v)$ and two edges from $H_{out}(w)$.

Having computed $H_G(v)$, we can construct a DAG called $D(G)$ containing $O(m + n \cdot logn)$ vertices and a mapping from vertices $v \in G$ to $h(v) \in D(G)$. Each vertice in $D(G)$ corresponds to an edge in $G - T$ and has out-degree at most 3. Furthermore, The vertices reachable in $D(G)$ from $h(v)$ form a 3-heap $H_G(v)$ in which the vertices of the heap correspond to edges of $G - T$ with tails on the path in T from v to t, in heap order by the values of $\delta(\varepsilon)$. The graph $D(G)$ provides a structure $H(s)$ representing the paths differing from the original shortest path by the addition of a single edge in $G - T$.

The Path Graph . Our goal is to produce a graph which can represent all $s - t$ paths, not just those paths with a single edge in $G - T$. For that reason, path graph $P(G)$ is defined and consists of the vertices of $D(G)$ with one additional vertex, the root $r = r(s)$ that is connected to $h(s)$ by an edge with $\delta(h(s))$. The vertices of $D(G)$ are the same in $P(G)$ but they are not weighted. The edges are given lengths. Then for each directed edge $(u, v) \in D(G)$ the corresponding edges in $P(G)$ are created and weighted by $\delta(v) - \delta(u)$. They are called Heap Edges. Then for each vertex $v \in P(G)$, which represents an edge not in a shortest path connecting a pair of vertices u and w, "cross edges" are created from v to $h(w)$ in $P(G)$ having a length $\delta(h(w))$. Every vertex in $P(G)$ only has an outgoing degree of 4 max. $P(G)'s$ paths starting from r are supposed to be a one-to-one length correspondence between $s - t$ paths in G.

In the end a new heap ordered $4 - Heap\ H(G)$ is build. Each vertex corresponds to a path in $P(G)$ rooted at r. The parent of any vertex has one fewer edge. The weight of a vertex is the length of the corresponding path.

Finding k-shortest paths . It is known that we can find the k smallest weight vertices in any heap in time $O(k)$ and that given the heap, there is a data structure that will output the vertices in order by weight, taking time $O(logi)$ to output the i_{th} vertex. So, having constructed the $P(G)$, we can find the k shortest s-t paths in G in time $O(k)$. The running time of Eppstein's algorithm is $O(m + n \cdot logn + k)$.

3.2.2 Loopless k-shortest paths *(by Yen)*

Yen raised an algorithm to solve the finding of k shortest paths without loops which is the basis of many other algorithms that are widely used for this problem.

Preliminaries .
Let $G(V, E)$ be a graph where :
- $u_1, u_2, u_3, u_4, \dots, u_N$: the nodes of the graph with u_1 the origin and u_N the sink
- $P = \langle u_1, u_i, \dots, u_j \rangle$, $1 \neq i \neq \dots \neq j$: the path from u_1 to u_j, passing through node u_i
- $d(u_i, u_j) \underset{>}{\overset{\leq}{}} 0, i \neq j$: the distance of the direct arc from u_i to u_j – if this arc exists, d is a finite number, otherwise, d is considered equal to infinity
- $P^k = \langle u_1, u_2^k, u_3^k, \dots, u_q^k, u_N \rangle, k = 1,2, \dots, K$: the k_{th} shortest path from u_1 to u_N, where $u_2^k, u_3^k, \dots, u_q^k$ are respectively the $2_{nd}, 3_{rd}, \dots, q_{th}$ node of th k_{th} shortest path
- $P_i^k, i = 1,2, \dots, q$: set of "deviations from path P^{k-1} at node u_i - a deviation from P^{k-1} at node u_i is the shortest of the paths that coincide with P^{k-1} from node u_i to the i_{th} node on the path and then deviate to a node that is different from any of the $(i + 1)_{st}$ nodes of those $P^j, j = 1,2, \dots, k - 1$, that have the same paths from u_1 to the u_i node as does P^{k-1}; and finally reaches u_N by a shortest subpath without passing any node that is already included in the first part of the path. Note that P_i^k is loopless and contains the same node no more than once
- R_i^k : root of P_i^k is the subpath of P_i^k that coincides with P^{k-1}, i.e., $u_1 - u_2^k - \dots - u_i^k$ in P_i^k
- S_i^k : spur of P_i^k is the last part of P_i^k that has only one node conciding with P^{k-1}, i.e., $u_i^k - \dots - u_N$ in P_i^k

Steps of the algorithm .
The Yen's algorithm that finds the K-shortest-path is as follows :

Iteration 1 . Determine P^1 by an efficient shortest-path algorithm (see [Yen70] for more details). Note that Yen's algorithm is an algorithm which finds the lengths of all shortest paths from a fixed node to all other nodes in an $N - node$ nonnegative-distance network ([Yen70]).

Iteration k $(k = 1,2,...,K)$. Determine P^k (in order to find P^k, the shortest paths P^1, P^2, ..., P^{k-1} must have been previously determined) as follows :

For each of the $i = 1, 2, ..., q$ nodes of the P^{k-1} shortest path, do the following

    a. Check if the subpath consisting of the first i nodes of P^{k-1} in sequence coincide with the subpath consisting of the first i nodes of P^j in sequence for j = 1, 2, ... , $k-1$. If so, set $d(u_i, u_w)$ where u_w is the $(i + 1)st$ node of P^j otherwise, make no changes. Then go to Step (b).
Note that distances $d(u_i, u_w)$ are set to infinity for computations in iteration k only. They should be replaced by their original values before iteration $k + 1$ starts.

    b. Apply a shortest-path algorithm to find the shortest path from u_i to u_N allowing it to pass through those nodes that are not yet included in the path. Note that the subpath from u_1 to u_i is defined as root of P_i^k, R_i^k, and the subpath from u_i to u_N is defined as spur of P_i^k, S_i^k.

    c. Find P_i^k by joining R_i^k and S_i^k. Then add P_i^k to List B. Note that it is necessary to store only the $K - k + 1$ shortest paths P_i^k in List B.

Find from List B the path(s) that have the minimum length. If the path(s) found plus the path(s) already in List A exceed K, the problem of k-shortest paths is solved. Otherwise, this path is denoted by P^k and move it from List B to List A - leaving alone the rest of the paths in List B. Then algorithm continues on iteration k + 1.

The above algorithm is developed from the fact that P^k is a deviation from $P^j, j = 1,2,...,$ $k-1$. More precisely, P^k must coincide with $P^j, j = 1,2,...,k-1$ for the first $m \geq 1$ nodes. Then, it deviates to a different node and finally arrives at the sink without passing each node more than once. Therefore, to obtain P^k, it is only necessary to look for all shortest deviations P^j and then choose the one with the shortest length.

To sum up, in each iteration k, step (a) of the approach sets $d(u_i, u_w) = \infty$ to force P^{k-1} to deviate at each node on the path without allowing the deviations to take any path that its length is shorter than P^{k-1}. This is followed by steps (b) and (c) that find the shortest deviations of P^{k-1}. Finally , the P^k is selected from all possible candidates in List B.

3.2.3 Discussion

Although, k-shortest-paths is a widely used method, the routes computed are very similar to each other and are not even considered as distinct to humans. Computing all shortest paths up to a number k produces many paths that are almost equal, sub-optimal and as result, only after a large number of k (1000 first shortest paths) the results tend to be satisfactory for the drivers. Consider, for example, the following situation : there are two long different highways from s to

t, where the travel time between them differs only for 5 minutes. To reach the highways, we need to drive through the city. For the number of different paths through the city to the faster highway, we have a combinatorial explosion. The number of different paths is exponential in the number of nodes and edges in the city, as we can independently combine short detours around a block, in the city. That means, it is not feasible to compute all shortest paths until we discover the alternative path on the slightly longer highway. Consequently, this method is quite impractical for computing alternatives.

3.3 Disjoint paths

The problem of finding link/node-disjoint paths between a pair of nodes in a network has received much attention in the past due to its theoretical as well as practical significance to many applications. Paths between a given pair of source and destination nodes in a network are called link disjoint if they have no common (i.e. overlapping) links, and node disjoint if, besides the source and destination nodes, they have no common nodes.

In this section, we focus on computing link-disjoint paths. In general, a link-disjoint paths algorithm can be extended to a node-disjoint algorithm with the concept of node splitting, i. e. replacing one node with two nodes that are linked together by a link with zero weights.

3.3.1 Disjoint paths algorithms

Link-disjoint path pair (LPP) problem . Given a $G(V, E)$, for a source-destination pair (s, t), find a set of two paths P_1 and P_2, such that $P_1 \cap P_2$ and the total length $l(P_1) + l(P_2)$ is minimized.

An intuitive method to determine two shortest link-disjoint paths between a pair of source and destination nodes consists of two steps. The first step retrieves the shortest path between a given pair of nodes in a graph. The second step is to remove all the links of that path from the graph, and to find the shortest path in the pruned graph. This method is referred as the remove-find (RF) method. Although the RF method is direct and simple, it has at least two disadvantages due to the removal of links belonging to the first shortest path. Provided that two link-disjoint paths exist, there is no guarantee that they will be found as illustrated in figure 3.3. The second link-disjoint shortest path may have a significantly larger length.

To surmount these problems, other methods have been devised to find shortest link-disjoint paths with minimal total length. Suurballe (see [Suur74]) proposed an algorithm, to find K node-disjoint paths with minimal total length using the path augmentation method. The path augmentation method is originally used to increase the size of a matching with an augmenting path [Die97] and to find a maximum flow or a minimum cost flow in a network ([FF62], [PS82]). The problem to find link/node disjoint paths can be viewed as a special case of the minimum

cost flow problem as demonstrated in [ST84], [Bha94], [Suu74]. The basic idea of Suurballe's algorithm is to construct a solution set of two disjoint paths based on the shortest path and a shortest augmenting path. K disjoint paths can be obtained by augmenting the $K - 1$ optimal disjoint paths with this algorithm. In 1994, Bhandari [Bah94] proposed an algorithm to find a pair of span-disjoint paths. The disjoint paths algorithm used by Bhandari is a modified version of Suurballe's algorithm [Suu74] that requires a special link weight transformation to facilitate the use of Dijkstra's. Bhandari made a simplification to Suurballe's algorithm by directly setting all the link weights on the first shortest path negative.

A simplified variant of Bhandari's Algorithm [Bah94], referred to as LBA (link disjoint version of Bhandari's algorithm), which can produce an optimal solution for the LPP problem is presented. Bhandari's algorithm is modified into a link-disjoint path pair algorithm LBA by omitting the node-splitting operation that ensures the node-disjointness and the graph transformations that ensure span-disjointness.

Preliminaries .

Before explaining the operation of LBA, some notations are introduced. Let $G(V,E)$ be a directed graph and (s,t) a source-destination pair. If the direction of a link is reversed, then its weight becomes negative, i.e. $w(u,v) = -w(v,u)$. Thus, if a path P from s to t exists, reversing the direction and the weight of all of its links, we will have a path directed from t to s, denoted by $\overleftarrow{P}$. Furthermore, $l(\overleftarrow{P}) = -l(P)$. A set, which consists of P_1 links whose reversed links appear on P_2 and vice versa, is denoted as $P_1 \widetilde{\cap} P_2 = \{(u \to v) \text{ and } (v \to u)|(u \to v) \in P_1 \text{ and } (v \to u) \in P_2\}$. In the following figures, bold lines represent edges on the shortest path(s) in the graph or its corresponding modified graph, dashed lines represent reversed edges which do not exist in the original graph and bold dashed lines represent such reversed links that appear on the shortest path.

Steps of the algorithm .

The steps of the LBA method are as follows :

- Find the shortest path P_{opt1} from node s to node t,
- Replace P_{opt1} with $\overleftarrow{P}_{opt1}$, a modified graph $G(V,E')$ is created,
- Find a shortest path P_{opt2} from node s to node t in the modified graph $G(V,E')$; if P_{opt2} does not exist, then stop,
- Take the union of P_{opt1} and P_{opt2} , remove from the union the link set which consists of P_{opt1} links whose reversed links appear in P_{opt2} and vice versa, then group the remaining links into two paths P'_{opt1} and P'_{opt2},

  i.e. $P'_{opt1} \cup P'_{opt2} = (P_{opt1} \cup P_{opt2})(P_1 \widetilde{\cap} P_2)$

The above-mentioned steps of LBA will be explained with an example, illustrated in figure 3.1. Suppose that we are required to find a set of two shortest disjoint paths between u_1 and u_6. In

the start, the shortest path from u_1 to u_6 is found as $P_{opt1} = \langle u_1, u_3, u_4, u_6 \rangle$ with minimum length 4.
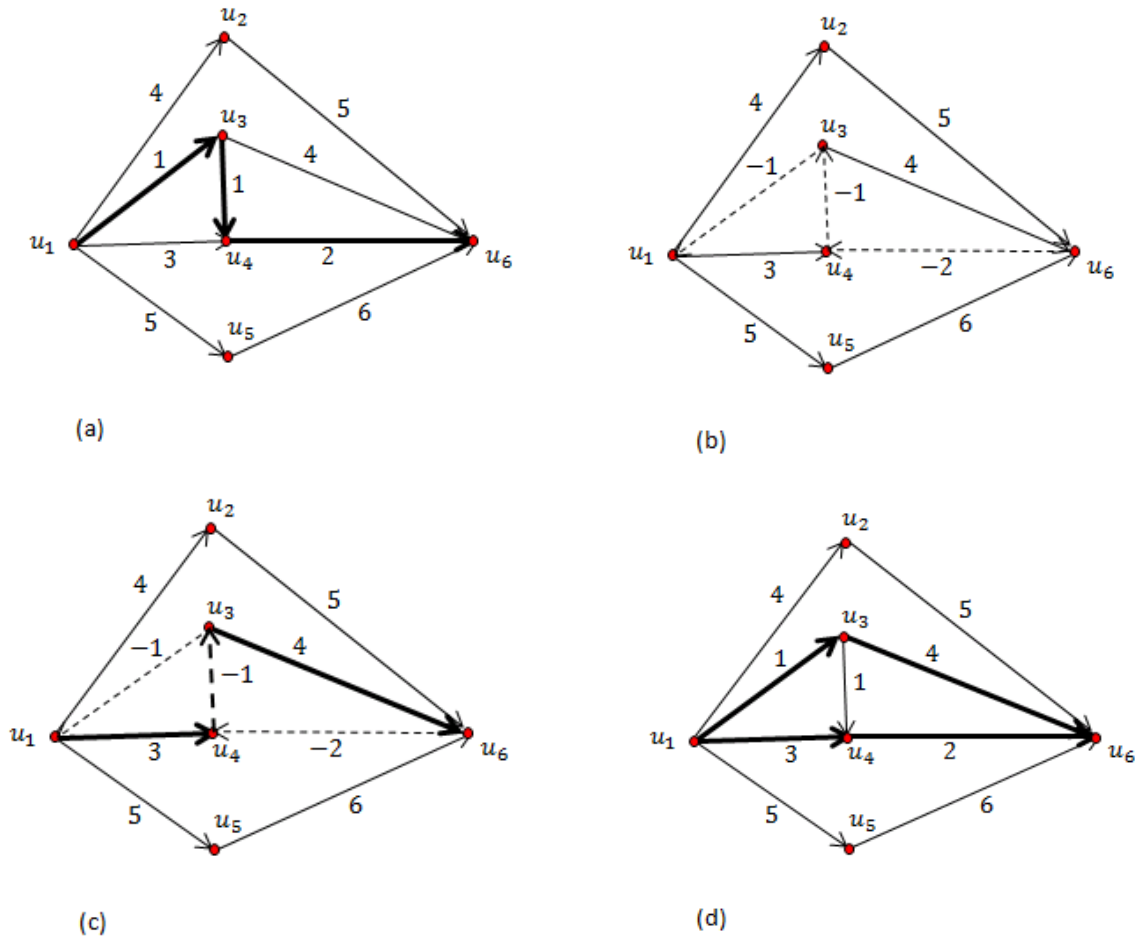
Figure 3.1 : (a) the first shortest path, (b) reversing direction and signs of shortest path, (c) the second shortest path, (d) computing the final pair of disjoint paths inspired by [GKM03]

In the next step, a modified graph $G(V, E')$ is created by reversing the direction and the sign of the weight of each edge on P_{opt1}. For instance, the edge $u_3 \rightarrow u_4$ with weight $w(u_3, u_4) = 1$ is replaced by the reverse edge $u_4 \rightarrow u_3$ with weight $w(u_4, u_3) = -w(u_3, u_4) = -1$. In step 3, the shortest path in the modified graph $P_{opt2} = \langle u_1, u_4, u_3, u_6 \rangle$ has length 6. In step 4, $P_{opt1} \widetilde{\cap} P_{opt2} = \{u_3 \rightarrow u_4, u_4 \rightarrow u_3\}$ is removed from the union $P_{opt1} \cup P_{opt2}$. The solution set of disjoint paths $\{P'_{opt1}, P'_{opt2}\} = \{\langle u_1, u_4, u_6 \rangle, \langle u_1, u_3, u_6 \rangle\}$ is obtained. The total length of this path set equals $5 + 5 = 10$, which is exactly the minimal total length of two link-disjoint paths in this graph.

For comparison, in figure 3.2, RF method is applied on the same topology with the same requirements. In step 1, the shortest path $\langle u_1, u_3, u_4, u_6 \rangle$ is retrieved. In the next step, a modified graph is created by removing all the edges of the shortest path. In step 3, the new

shortest path $\langle u_1, u_5, u_6 \rangle$ is computed. Thus, the set $\{\langle u_1, u_3, u_4, u_6 \rangle , \langle u_1, u_5, u_6 \rangle\}$ has a total length $4 + 11 = 15$, which is bigger than 10 as found with LBA. This example illustrates that the RF method cannot guarantee to find the optimal solution. More important, in the graph shown in figure 3.3(a), although there exist two link-disjoint paths between u_1 and u_4, RF cannot find the second path in step 2 as shown in figure 3.3(b). LBA, on the other hand, still returns the optimal set in this case.
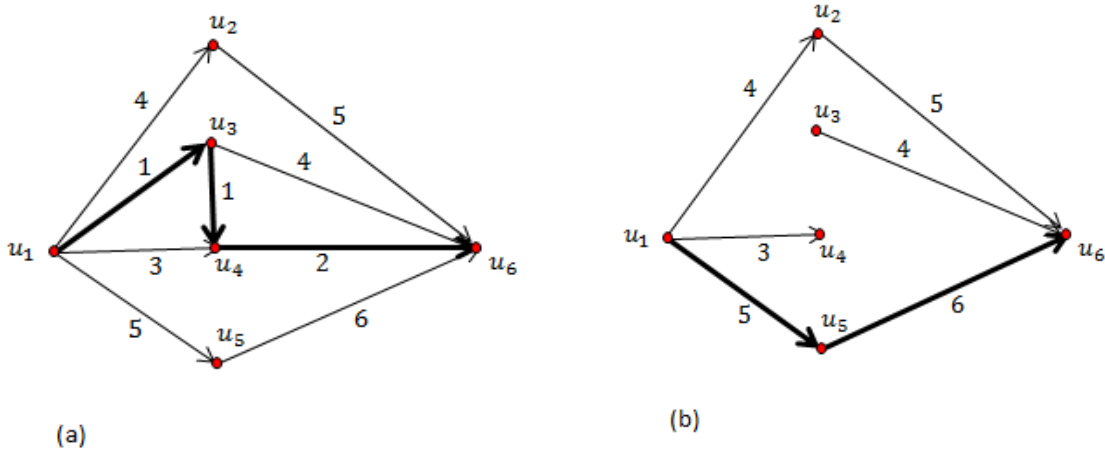
Figure 3.2 : RF method, (a) the shortest path, (b) the second shortest path after the removal of the edges of the first shortest path. Figures (a),(b) were inspired by [GKM03]
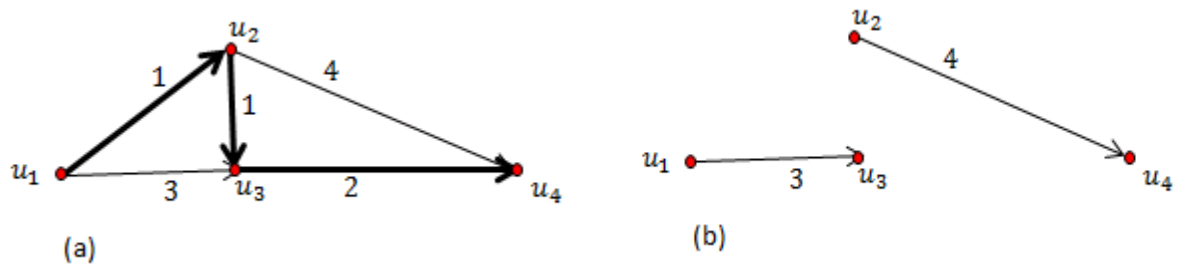
Figure 3.3 : the disadvantages of RF method inspired by [GKM03], (a) the first shortest path, (b) although in the graph there are 2 disjoint paths, RF, due to the removal of edges cannot find the second path

3.3.2 Discussion

The disjoint-paths method is a way to find alternative routes. However, the results obtained can be occasionally not at all satisfactory. If RF method is applied, it is most likely that the resulting

alternatives (apart from the shortest path) have large lengths, since no edge is allowed to be common. Moreover, this method cannot guarantee that existing alternatives in the graph will actually be found. On the other hand, LBA method certainly achieves better results than RF. However, it is not that helpful for finding good alternative routes in road networks, since some road segments may be shared between several of the diverse routes that we would like to find, which are therefore not disjoint. We usually do not request total disjointness for alternatives routes in road planning. Thus, this method is not that proper for our problem. Instead, it can be useful in cases that link-disjoint path problems actually occur. Such kind of problem usually appears in computer network design where aspects as survivability, load balancing and network resource utilization are strived for.

3.4 Pareto optimality *(with SHARC)*

Pareto optimality or Pareto efficiency is a concept originally appeared in economics and then applied in engineering. In engineering, given a set of choices (i.e. alternative routes in a map) and a way of valuing them, the Pareto set is the set of choices that are Pareto optimal. An outcome of a game is Pareto optimal if there is no other outcome that makes every player at least as well off and at least one player strictly better off. That is to say, a Pareto optimal outcome cannot be improved upon without hurting at least one player.

3.4.1 Pareto optimality algorithm

Introducing the concept in the field of routing, Pareto optimality is a classical approach to compute alternatives. In general, up to a short time ago, routing on road networks focused on single-criteria scenarios. The goal was to find the quickest route within a network. However, the quickest route is often not the best one. A user might be willing to accept slightly longer travel times if the cost of the journey is less. A common approach to cope with such a situation is to find Pareto-optimal routes, concerning a variety of metrics. Specifically, we can consider several weight functions for the edges like travel time, distance, fuel consumption or scenic value and compute pareto-optimal alternative routes in a way that each route is better than any other route with respect to at least one metric under consideration, e.g. fuel consumption. Thus, the concept of multi-criteria search in road networks is introduced.

The straightforward approach to find all Pareto optimal paths is the generalization ([Han79], [Mar84]) of Dijkstra's algorithm: Each node $v \in V$ gets a number of multi-dimensional labels assigned, representing all Pareto paths to v. For the bicriteria case, [Han79] was the first presenting such a generalization, while [Mar84] describes multi-criteria algorithms in detail. By this generalization, Dijkstra loses the label-setting property, i.e., now a node may be visited more than once. It turns out that a crucial problem for multi-criteria routing is the number of labels assigned to the nodes. The more labels are created, the more nodes are reinserted in the

priority queue yielding considerably slowdowns compared to the single-criteria setup. In the worst case, the number of labels can be exponential in $|V|$ yielding impractical running times [Han79].

In this section, an efficient speed-up technique for multi-criteria routing is presented. This speed-up technique is an augmented version of SHARC which is a combination of two speed-up techniques presented in the chapter 2, contraction (Shortcuts) and arc flags(ARC). By augmenting the main subroutines of SHARC to multi-criteria variants and by changing the intuition when setting Arc-Flags, a very efficient method for the multi-criteria scenario is generated (see [DW09]).

Preliminaries .

Let $G(V, E)$ be a directed graph. The main difference between single and multi criteria routing is that the labels assigned to edges contain more than one weight and in particular they are vectors in R_+^k . Let $L = (w_1, w_2, \ldots, w_k)$ and $L' = (w_1', w_2', \ldots, w_k')$ be two labels. L dominates another label L' if $w_i < w_i'$ holds for one $1 \leq i \leq k$ and $w_i \leq w_i'$ holds for each $1 \leq j \leq k$. The sum of L and L' is defined by $L \oplus L' = (w_1 + w_1', w_2 + w_2', \ldots, w_k + w_k')$. The minimum and maximum component of L is defined by $\underline{L} = \min_{i \leq i \leq k} w_i$ and $\overline{L} = \max_{i \leq i \leq k} w_i$ respectively. Finally, the length $d(s, t)$ of an $s - t$ path $P = \langle e_1, e_2, \ldots, e_r \rangle$ is given by $L(e_1) \oplus L(e_2) \oplus \ldots \oplus L(e_r)$. In contrast to a single-criteria scenario, many paths exist between two nodes that do not dominate each other. In this work, we are interested in the Pareto-set $D(s, t) = \{d_1(s, t), d_2(s, t), \ldots, d_x(s, t)\}$ consisting of all non-dominated path lengths $d_i(s, t)$ between s and t. $|D(s, t)|$ is called size of a Pareto-set. Note that by storing a predecessor for each d_i, Pareto paths can be computed as well. The len function assigns a label L in each edge.

Steps of the algorithm .

Augmenting Ingredients . In this section, we show how to augment Dijkstra-search, arc flags and contraction in order to guarantee correctness in a multi-criteria scenario ([DD09]).

Dijkstra search . Computing a Pareto set $D(s, t)$ can be done by a straightforward generalization of Dijkstra ([Han79], [Mar84]). For managing the different distance-vector at each node v, a list of labels $list(v)$ is maintained.
- Initialize the source node s with a label $d(s, s) = (0, \ldots, 0)$ and any other list as empty.
- Insert $d(s, s)$ to a priority queue

In each iteration step,
- Extract the label with the smallest minimum component
- For all outgoing edges (u, v), a temporary label $d(s, v) = d(s, u) + len(u, v)$ is created
- If $d(s, v)$ is not dominated by any of the labels in $list(v)$,
    - add $d(s, v)$ to $list(v)$

- remove all labels from $list(v)$ that are dominated by $d(s, v)$
- Stop the query as soon as $list(t) \neq \emptyset$ and all labels in the priority queue are dominated by all labels in $list(t)$

Pareto Path Graph is a graph (PPG) constructed by computing $D(s, u)$ for a given source s and all nodes $u \in V$. An edge (u, v) is a PPG-edge if and only if it is a part of at least one Pareto-optimal path from s to v.

Arc flags . In a single-criteria scenario, an arc-flag $F_e(R_i)$ denotes whether e has to be considered for a shortest-path query targeting a node within the region R_i . In other words, the flag is set if e is important for (at least one target node) in R_i. The adaption to multi-criteria states that an arc-flag $F_e(R_i)$ is set to true, if e is important for at least one Pareto path targeting a node in R_i. For the "augmented" computation of arc-flags, we build a pareto path graph (PGG) in $\overleftarrow{G}$ for all boundary nodes $b \in B_{R_i}$ of all regions R_i . Then, we set $F_{(u,v)}(R_i) = 1$ if (u, v) is a PGG-edge for at least one PPG grown from all boundary nodes $b \in B_{R_i}$. SHARC is based on multi-level Arc-Flags. Hence, we grow a PPG in $\overleftarrow{G}$ for all boundary nodes b on the lower level and stop the growth as soon as all labels attached to the nodes in the superregion of R_i dominate all labels in the priority queue. Then, we set an arc-flag to true if the edge is a PPG edge of at least one Pareto path graph (see [DD09]).

Contraction . The Pareto contraction routine is similar to the one mentioned in chapter 2. First, the unimportant nodes are contracted and in order to preserve Pareto paths between non-removed edges, new edges called shortcuts are added to the graph. Then, edge-reduction step is applied to remove unneeded shortcuts.

SHARC query . Augmenting the SHARC-query is straightforward. For computing a Pareto-set $D(s, t)$, the modified multi-criteria Dijkstra explained before is applied on the output graph. The modifications are then the same as for the single-criteria variant of SHARC: When settling a node n, we compute the lowest level i on which n and the target node t are in the same super-regions. Moreover, we consider only those edges outgoing from n having a set arc-flag on level i for the corresponding region of t. In other words, we prune edges that are not important for the current query.

The number of Pareto-optimal paths can be quite large. So, the number of computed paths can be decreased by tightening the domination criteria to keep only paths that are sufficiently different. Therefore, the travel time metric is defined so as to be the dominating metric W. Then, the tightened definition of dominance is as follows: Besides the constraints mentioned before, a label $L = (W, w_1, \dots, w_{k-1})$ dominates another label $L' = (W', w'_1, \dots, w'_{k-1})$ if $W \cdot (1 + \varepsilon) < W'$ holds. In other words, Pareto-paths are allowed if they are up to ε times longer (with respect to the dominating metric). Note that by this notion, this has to hold for all sub-paths as well.

3.4.2 Discussion

Pareto optimality method is a method which concept differentiates from the classical idea of alternative route finding of the previous algorithms. It introduces the concept of multi-criteria routing in large-scale networks. Thus, pareto optimality with SHARC is an efficient speed-up technique for computing multi-criteria paths in large-scale networks which results from the augmentation of single-criteria routines of SHARC to multi-criteria versions. The main problem that can arise is the large number of Pareto-routes, making preprocessing and query times impractical for large instances. This problem is dealt with pruning, as described above.

3.5 Penalty method

Obviously, the aforementioned algorithms are capable of generating a large number of alternative paths, which can be useful in a number of transport planning instances. However, many of these alternative paths are likely to share a large number of edges. If one can define a measure of dissimilarity between these paths, then a subset of these paths can be selected, so that the minimum dissimilarity is maximized. In fact, this description refers to the path dissimilarity problem (PDP) that is a routing problem in which a set of P paths from an origin to a destination must be generated with minimum length and maximum dissimilarity. One of the most relevant procedures developed to solve this problem is the Penalty Method.

The Penalty method was originally suggested in the context of hazardous materials routing by Johnson et al.(1992) and used by Ruphail et al. (1995), in a decision support system to generate economically different paths over a network characterized by time-dependent link travel times.

3.5.1 Penalty algorithm

Steps of the algorithm .
The Penalty Method is based on a repetitive application of an appropriate shortest path algorithm. The basic steps of the algorithm are described below.

- Compute a shortest path with a shortest path algorithm, i.e. Dijkstra
- Add it to the solution set or alternative route subgraph,
- Increase the edge weights on this path and start from the beginning until we are satisfied with the solution

Hence, the repeated selection of the same set of links is discouraged and dissimilar paths may be generated as results. However, the paths generated may not be completely different as some subpaths may still be shorter than a full detour (depending on the

increase). Thus, there are several choices or "dimensions" that one have to take into account for the implementation ([AEB00]). Specifically,

- **Penalized units .** Penalties can be applied to the links, or nodes, or both.
- **Penalty structure .** An additive penalty (i.e. adding a fixed positive amount to the impedance), or a multiplicative penalty structure (i.e. multiplying the current impedance by a factor greater than one) can be used. If one uses a multiplicative penalty formula, the new impedance can be based on the current impedance (which may have been penalized before), or on the original impedance.
- **Penalty magnitude .** If a relatively large penalty is chosen, then links that appear in generated paths are discouraged more heavily. Smaller penalties, on the other hand allow for more frequent appearances of links in generated paths.
- **Penalized paths .** Penalties can be applied to the most-recently found path only, or to all paths found so far or to paths "close" to the most-recently found.

The above choices lead to various experiments with different penalty mechanisms. For example, as far as penalty magnitude is concerned, the higher the penalty, the more the new shortest path deviates from the last one. Although a small penalty may not achieve the goal of dissimilarity, a large penalty may eliminate possible good viable paths. Consider the following case. There are two good alternatives whose first part is common, while the second is different. The algorithm computes the first route and penalizes its edges. If the penalty has small magnitude, the algorithm will find the second alternative with higher probability since the total length of the second alternative will remain relatively small, despite the increase in the edges of the first common part. On the contrary, if the penalty is very high, then it is likely that the second alternative cannot be found or a large detour will be computed for the first part instead, as illustrated in figure 3.4.
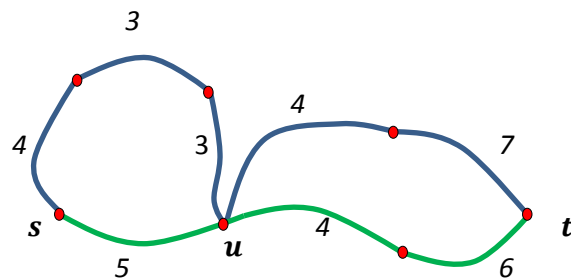
(a) Shortest path between s-t illustrated by green

(b1) Penalization with small penalty factor 0.2

(c1)New shortest path illustrated by orange

(b2) Penalization with large penalty factor 1.1

(c2) New shortest path illustrated by orange
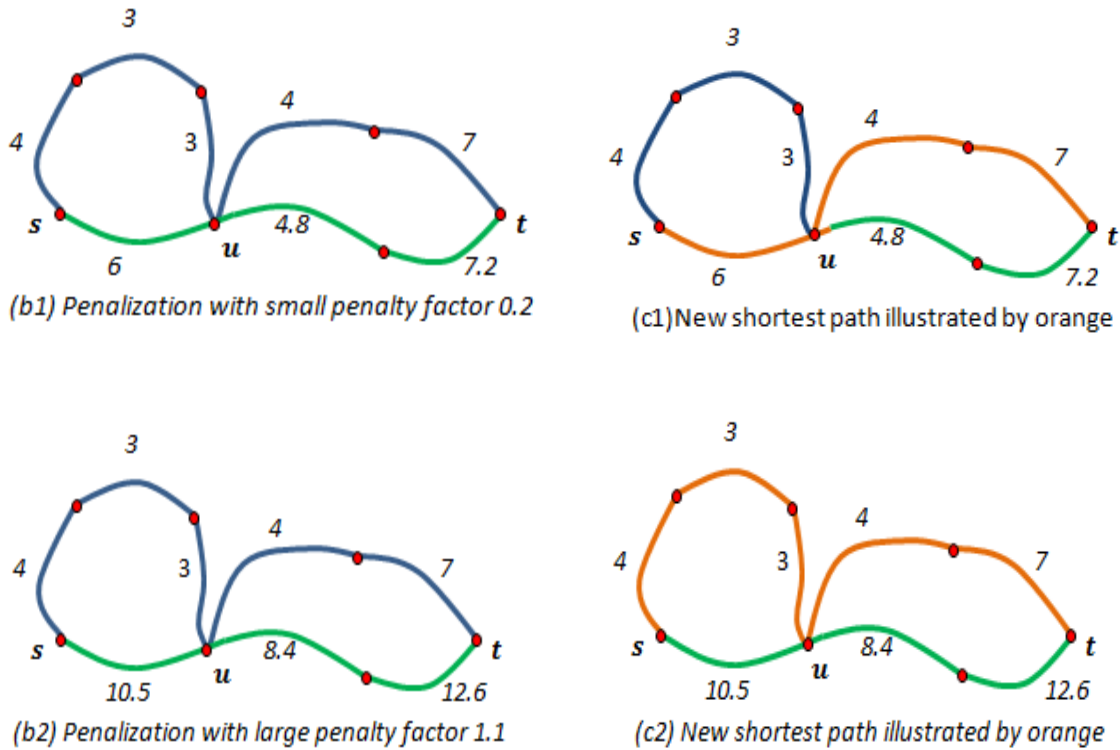
Figure 3.4 : Graph with 2 good alternative routes. The first part of the alternatives $s - u$ with weight 5 should be common because there are not meaningful alternatives. The second part of the alternatives $u - t$ is different as illustrated in figure (a). Figures (b1), (b2) show the graph after the penalization with small and large factor respectively. Figures (c1), (c2) show the second alternative that the algorithm computes after the penalization. When the factor is too big, the results are not the desirable ones

Furthermore, as far as penalty structure is concerned, one can add an absolute value on each edge of the shortest path, but this depends on the assembly and the structure of the graph and penalizes short paths with many edges. To by-pass this, a multiplicative penalty structure can be used. In other words, edges are penalized by a fraction "penalty-factor" of the initial edge weight to the weight of the edge. Like before, although the highest the penalty-factor is, the more disjoint are the paths found, a similar problem still appears. For instance, the first part of the route has no meaningful alternative but the second part has 5. That means that the first part of the route is likely to be increased several times during the iterations (multiple-increase). In this case, we can get a shortest path with a very long detour on the first part of the route. To circumvent this problem, the number of increases of a single edge can be limited. When a new shortest path does not increase the weight of at least one edge, it is an indication of a natural saturation of the number of available alternatives.

Another problem that rises is that depending on the "penalized paths" strategy, a new alternative can have many small detours compared to the last alternative. For example, the last computed path is a long motorway and the new shortest path is almost equal to the last one, but at the middle of the motorway, it contains a very short detour from the long motorway on a less important road (due to the increase). There can occur many of those small hops that look unpleasant for humans and are not considered as distinct alternatives. To tackle this problem, the concept of penalization strategy arises. In particular, instead of increasing only the weights of the currently computed shortest path, the weights of the edges around the path (i.e. in a radius r from the path) can be also increased. This avoids small hops, as edges on potential hops are probably not shorter. Still, potential, meaningful alternatives close to the current path can be unfairly penalized. To avoid this, one can increase only the weights of the edges, which leave and join edges of the current alternative route subgraph. This strategy is called rejoin-penalty. It should be additive and dependent on the general penalty factor k and the distance from s to t, e.g. $rejoin - penalty \in [0 \dots (penalty - factor) \cdot 0.5 \cdot d(s, t)]$([BDGS11]).

3.5.2 Discussion

Penalty method is another way of finding alternatives. However, as we described above, it can produce alternatives not admissible, since the various parameters should be chosen carefully. Intuitively, we believe that it can be better used as a complement to other methods, as we will soon see in chapter 4.

3.6 Conclusion

In this chapter, various methods in the direction of alternative route planning were presented. Both of them have a specific methodology in identifying the alternative paths, leading either to good or bad results. Most of them, however, usually produce bad results because of the way they find alternatives (requiring total disjointness, minimum lengths, introducing large penalties etc.) and as a result they are suitable only in specific circumstances. Thus, in chapter 4, we introduce a new method in the field of alternative route planning, called plateau method, which is suitable for identifying alternatives in road networks due to the strategy followed. Before that, we introduce the concept of admissible alternative routes. Aside from how well each algorithm discovers routes, we want alternative paths that are meaningful to each driver and as a result satisfy a set of characteristics/criteria. These criteria will be incorporated in plateau method and the other approaches and they will filter candidate routes so as to hold the best of them.

Chapter 4

PLATEAU METHOD AND ADMISSIBILITY CRITERIA

In the previous chapter, we presented methods for finding alternative routes which are widely used in several scientific fields. A well-known approach is the k-shortest paths algorithm, but it is impractical because a reasonable alternative in a road network is probably not among the first few thousand paths. Besides, disjoint paths is another approach presented, but the requirement for totally nonoverlapping paths can lead to the loss of some potentially good alternatives. Thus, we do not want only to calculate alternative routes, but compute reasonable alternatives which are considered as distinct to the user. For that reason, we define a subset of alternative routes, admissible alternative routes that meet certain desired characteristics. In this context, the best published results we are aware of are produced by the choice routing algorithm or plateau method which is presented in detail. The stages of the algorithm are presented in detail, but apart from the identification of the plateaux chains, include the evaluation of them based on admissibility criteria, goodness metric etc. For this evaluation, we also present heuristics in order to measure sharing, stretch etc. and an objective function in order to sort them and choose the best one to add in the alternative graph AG in every round ([ADGW10]).

4.1 Admissible alternative routes

As mentioned above, an alternative path P in a road network should satisfy certain properties in order to be reasonable and natural to the user. Obviously, an alternative route must be substantially different from the optimal path and must not be much longer. But this is not enough. Alternative routes should avoid unnecessary detours. Formally, they must be locally optimal, that is to say, every subpath up to a certain length must be shortest path. Thus, path P belonging in the class of admissible alternative routes, should have (see also [ADGW10]) :

- Limited sharing
- Bounded stretch, even for all subpaths
- Local Optimality

Definitions . Let $G(V, E)$ be a directed graph with nonnegative, integral weights on the edges. Given any path P in G, $|P|$ is its number of edges and $l(P)$ the sum of the weights of its edges. By extension, $l(P \cap Q)$ is the sum of the lengths of the edges shared by paths P and Q, and $l(P \backslash Q)$ is equal to $l(P) - l(P \cap Q)$.

Sharing . The idea of sharing refers to the amount of difference between the alternative routes P and the optimal route P_{opt} computed by a shortest path algorithm, i.e. Dijkstra. The total length of the edges they share must be a small fraction of $l(P_{opt})$.

(a) Limited sharing   (b) High sharing
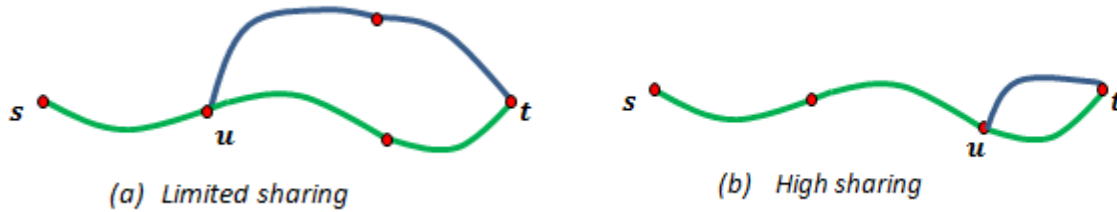
Figure 4.1 : Graphical representation of sharing. Shortest path is illustrated by green and the alternative by blue. (a) The length of the alternative is a small fraction of (P_{opt}) , in contrast to (b) where the alternative slightly differs from the optimal route.

Stretch . Stretch refers to the length of the path between two points on the alternative route. A path P has $(1 + \varepsilon)$ uniformly bounded stretch $((1 + \varepsilon) - UBS$ if every subpath (including P itself) has stretch at most $(1 + \varepsilon)$, i.e. P is at most $(1 + \varepsilon)$ times larger than the optimal path P_{opt} .

(a) large stretch   (b) bounded stretch
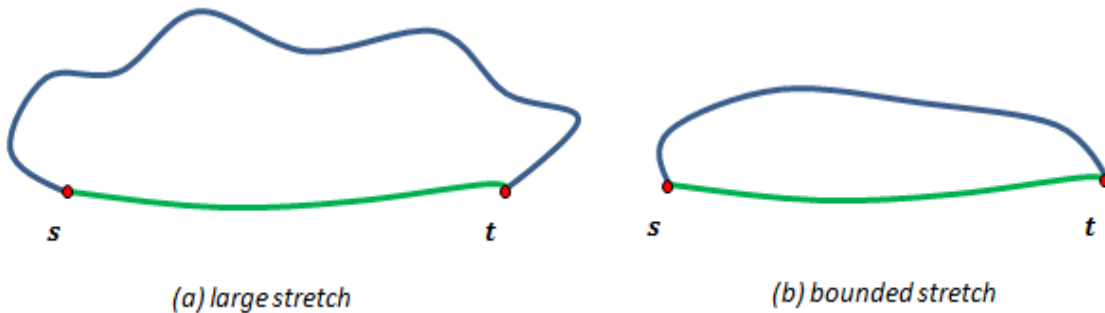
Figure 4.2 : Graphical representation of stretch. Shortest path is depicted by green and the alternative by blue. In the first case, (a), the alternative is much longer than the optimal route. In contrast, in the second case, (b), the alternative route is a bit longer than the optimal path.

Local Optimality . The idea of local optimality refers to the lack of unnecessary detours in the alternative routes. Every subpath of the alternative route up to certain length should be optimal. While driving along it, every local decision must make sense. To formalize this notion, a path P is T locally optimal $(T - LO)$ if every subpath P' of P with $l(P') \leq T$ is a shortest path. Besides, since the path P is not continuous, but discrete, a second condition must be true. If P' is a subpath of P with $l(P') > T$ and $l(P'') < T$, where P'' is the path obtained by removing the endpoints of P', then P' must be a shortest path. Note that a path that is not locally optimal includes a local detour.
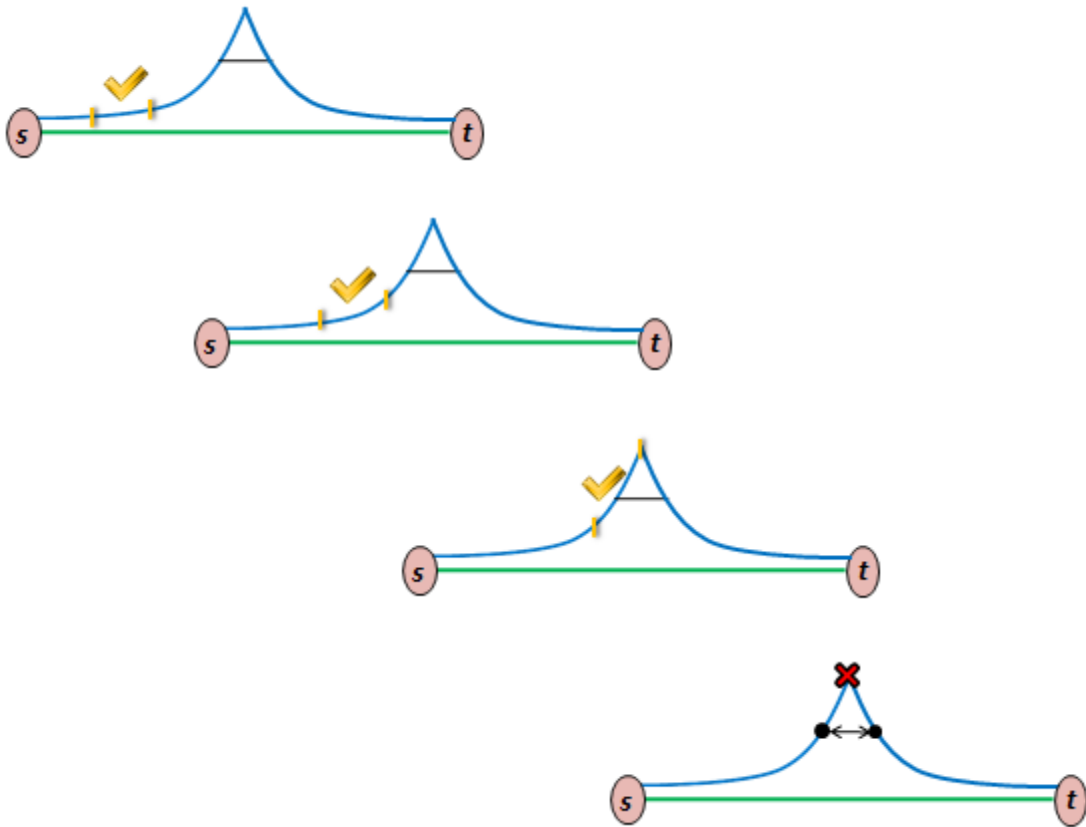
Figure 4.3 : Graphical representation of local optimality. The three first figures illustrate subpaths that are T locally optimal, while the last figure shows the case of an unnecessary detour

Given these definitions, the class of admissible alternative routes is defined formally below. Given a shortest path P_{opt} between s and t, an $s - t$ path P is an admissible alternative is satisfies the following conditions :

Limited Sharing .

Let $\sigma(P) := l(P \cap P_{opt})$ be the sharing amount of a path P with the optimal path P_{opt} .

A path P has limited sharing, if $\sigma(P) \leq \gamma \cdot l(P_{opt})$, $0 \leq \gamma \leq 1$.

Note that for $\gamma = 0$, the paths are totally disjoint, while for $\gamma = 1$, the paths can be identical.

Uniformly Bounded Stretch .

A path P has bounded stretch, if $l(P) \leq (1 + \varepsilon) \cdot l(P_{opt})$.

Local Optimality .

A path P is T-locally optimal for $T = a \cdot l(P_{opt}), \quad 0 < a < 1$, if

- P' is a shortest path if $P' \subseteq P$ with $l(P') < T$
- P' is a shortest path if $P' \subseteq P, P'' \subseteq P'$ with $l(P') > T$ and $l(P'') < T$

Given the conditions that alternatives should obey, an algorithm can generate zero, one or multiple admissible alternatives, depending on the input and the choice of parameters. If there are multiple alternatives, we can sort them according to an objective function $f(\cdot)$, which may depend on any number of parameters, possibly including α, ε and γ. Generally, admissible paths with low sharing, low stretch and high local optimality are preferred.

4.2 Plateau method *(Choice routing algorithm)*

Having defined the admissible alternative routes, we can present a new approach for finding alternative routes which computes alternative routes that "naturally" meet admissibility criteria and especially local optimality, to greater extent than the methods presented in chapter 3. The related algorithm is called plateau method or choice routing algorithm and it was invented by Alan Henry Jones in 2009 (see also [Jon12]).

Plateau Method is one of the most significant algorithms for generating a plurality of diverse routes from a source to a destination in a graph. Such a method may be used for route planning and navigation in road networks, but may also be used in other applications where the costs can be described by a weighted graph and where there are no cycles of negative cost, such as routing of packets in computer network, finding paths for wiring in integrated circuits etc.

Consider that we have converted the road network into a graph $G(V, E)$ (figure 4.4), and we want to compute the alternative routes between a source node s and a target node t.

The method comprises some basic steps which are described below in high level :

- Generating a source routing tree from the source s
- Generating a destination routing tree to the destination t
- Combining the source and destination trees to form the alternative routes

The three steps will be presented analytically below. Note that the third step is the most important step and the basic element of the algorithm for computing the alternatives.
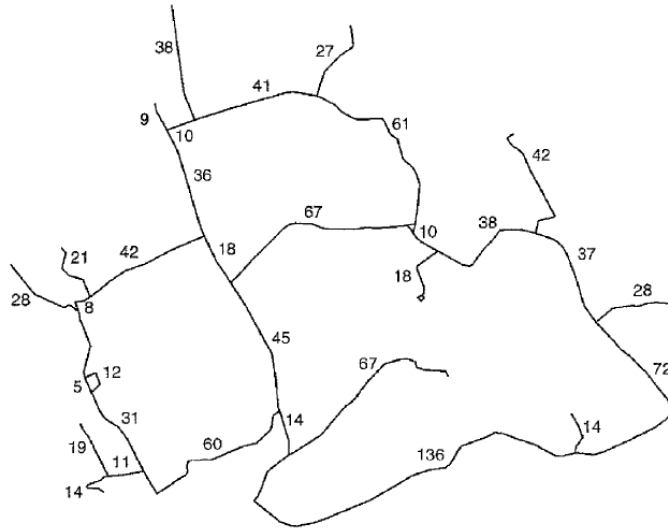
Figure 4.4 : A road network as a graph as given in [Jon12]. The weights on the edges are distances.

Source or Forward tree . The first step of the method is to compute the shortest path tree from the source node *s* to all other nodes. This is typically performed using Dijkstra algorithm or its variants, such as A* algorithm, often enhanced by clever use of trunk roads, precomputation and graph restrictions to speed up the computation or reduce the storage requirements (see also Chapter 2 for speed-up techniques). Figure 4.5 illustrates the resulting forward tree.
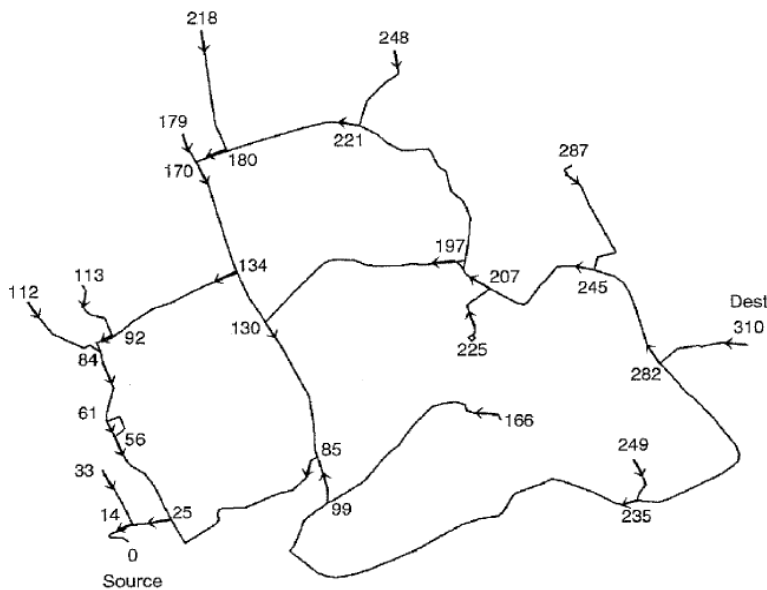
Figure 4.5 : The source tree from node s as given in [Jon12]

In figure 4.5, for each node of the graph(end of road segments), it is annotated the distance from the source node *s*. Moreover, in every node, there is only one outgoing arrowhead that

55

shows the way back to the source using the shortest path. In other words, the arrowhead shows the predecesessor of the current node in the shortest path from s. This arrowhead is called back pointer. Note that back pointers are in the opposite direction to the direction of travel. These are computed as a necessary part of the Dijkstra of A* algorithm (i.e. stored as pointers).

For example, the shortest path from the source to the destination node has length 310 and can be traced backwards by following the back pointers through the nodes whose distances are 282, 245, 207, 197, 130, 85, 25, 14 and finally 0.

Destination or Backward tree . The second step of the algorithm is to compute the shortest path tree to the destination node t from all other nodes. Instead of executing multiple Dijkstras, we can execute a single one from node t in the backward graph $\overleftarrow{G}$ (having reversed the edges or taking into account only the ingoing edges). So, this is just a variant of the previous algorithm and the output is shown in figure 4.6.
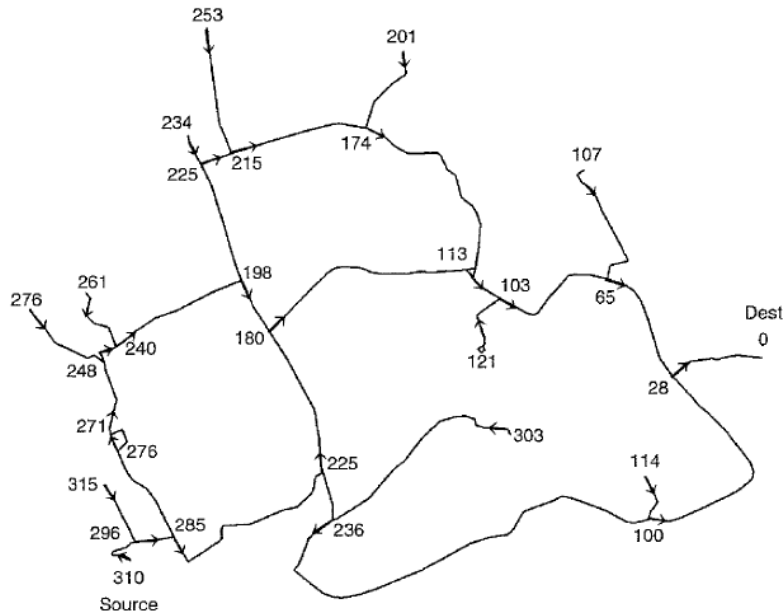
Figure 4.6 : The destination tree to node t, taken from [Jon12]

In figure 4.6, the annotations give the distances to the destination node t along the shortest path. The arrowheads show again, for each node, the way to the destination using the shortest path. Under another perspective, they show the successor (node) for each node in order to reach the destination along the shortest path. These arrowheads are called front-pointers.

For example, the shortest path to the destination from the node at the top left with distance 234 is found by following the front-pointers through the adjacent nodes whose distances are 225, 215, 174, 113, 103, 65, 28, 0. This tree also encodes the globally shortest path, which is found by following the arrowheads from the source node to the destination and it will always be identical to the one found in the source tree.

Note that the subtle difference between the trees, is that source tree encodes the shortest path routes from a single node to many others, while the destination tree encodes the shortest path routes to a single node from many others.

Combined Tree. The next step in plateau method is to sum up for each node its minimum distances from s and t, found in the forward and backward tree respectively. The resulting output is the combined tree.

Figure 4.7 : Combined Tree. Figure is taken from [Jon12]

For instance, the number 310 for the source node was arrived at by adding the corresponding numbers 0 and 310 from source and destination trees, as shown in figures 4.5 and 4.6. These numbers have a powerful interpretation. At any node v, they represent the cost/length of the shortest path route from source node s to the destination node t via the node v (we will refer to this distance as via node distance of node v). Therefore, we have computed the set of shortest paths routes P_v from source to destination via any node v in the graph. These paths P_v are formed as the concatenation of two shortest paths $s - v$ and $v - t$. Consequently, having computed the combined tree, we have managed to compute the via node distances of all the candidate alternative paths via any node in the graph (i.e. To go from s to t using the path via the node at the bottom, you cross distance equal to 335 units). However, the number of candidate paths is huge and some of them are irrelevant when planning our route from s to t. Thus, it is necessary we introduce another step in the method in order to reduce the candidate via paths. This step identifies routes that are meaningful, exploiting paths with the same via node distances. Specifically, one can observe in figure 4.8, that there are chains of adjacent nodes which have the same via node distance. Obviously, the nodes that lie on the shortest path

route from s to t have the same via node distance, which is exactly the length of the optimal route. However, there are other such chains illustrated on the figure below.

Figure 4.8 : Plateaux : chains of adjacent nodes with the same via node distance given in paper [Jon12]

Each of the maximal-length chains with the same via node distance is called *plateau*. This constant via node distance can be represented as the constant height of some plane-plateau.

Plateaux identification . A typical procedure for finding the plateaux is the following :

- we begin by giving each node a single bit that indicates if we have visited it or not. Initially, it is 0 for not-visited
- we scan every node in turn,
    - if it is visited (1), we move on to the next node in scan
    - if it is non-visited (0), we set the bit to 1 to indicate that it is visited. For such newly-marked node, call it q, we begin a list of adjacent nodes in the chain by adding just a reference to that node q. We then follow the back pointers in the source tree and for each node we meet, call it u_1, we check if its via node distance is the same as $q's$. If the via node distances are the same, we mark it as visited and add a reference to q to the list and repeat the procedure for the predecessor $u_2 = p(u_1)$. When this procedure finishes, that is to say, it was found a node with different via node distance form q or there was no other back-pointer(null), we return to node q, we follow the front-pointers and we execute the same steps for the successors. When this second procedure finishes, we have a list that comprises all of the nodes in the chain that node q is part of. This list represents a plateau. We continue to the next node in the scan.

A plateau is formed when the source and destination trees traverse a chain of road segments in the same directions. This indicates that the chain is both useful for getting away from the source and for getting towards the destination. Such chains tend to use the best roads in their vicinity and are aligned to help in getting from source to destination. To make a complete route out of a plateau, we simply have to follow the arrowheads in the source and destination trees from the endpoints of the plateau.

Figure 4.9 : Plateaux illustrated by bold lines in [Jon12]

Figure 4.9 shows the plateaux found. Suppose we check plateau between A and B nodes. For that plateau, its length l_p is simply the difference in the values at nodes A and B in the source tree ($221 - 170 = 51$ from figure 4.5) or in the destination tree ($225 - 174 = 51$ from figure 4.6). The shortest path route from source to plateau has length l_{sp} that is equal to the distance of node A from s. The shortest path route from plateau to destination has length l_{pt} that is equal to the distance of node B from t. Thus, the total length of the optimum route that incorporates the plateau is given by $l_{sp} + l_p + l_{pt}$. This value must be exactly the length of the shortest path route from source to destination via any one of the nodes in the plateau.

Generally, we want plateaux that are long. In other words, a useful plateau will tend to be longer than those that are less useful, as it indicates a long stretch of route that is fast and well-aligned compared to others in its vicinity. Thus, we are looking for plateaux with a larger value of l_p.

Moreover, a useful plateau will tend to be part of a route that is not too long, as we are not interested in long plateaux if they are found at a great distance from both source and destination. Thus, we are looking for plateaux with a smaller value of $l_{sp} + l_p + l_{pt}$. As a consequence, we compute a goodness factor for each plateau that takes into account the lengths of the different parts of the alternative.

Goodness . Goodness factor is a metric in order to evaluate and sort plateaux taking into account the different lengths of the alternative, l_p, l_{sp}, l_{pt} . Goodness is defined as the length of the plateau minus the length of the route that the plateau is a part of :

$$l_p - (l_{sp} + l_p + l_{pt}) = -(l_{sp} + l_{pt})$$

In order to make this measure independent of the length units, we divide it by the length of the globally shortest path route and this is called raw goodness :

$$raw\ goodness = RG = \frac{-(l_{sp} + l_{pt})}{l_{shortest\ path}}$$

Thus, plateaux with raw goodness closer to zero are better and more useful than others with smaller value. For example, from figure 4.9, we can compute the raw goodness of the different plateaux depicted.

Via node distance	l_{sp}	l_p	l_{pt}	$-(l_{sp} + l_{pt})$	Raw goodness
395	170	51	174	−344	**−1.11**
335	99	136	100	−199	**−0.64**
332	56	78	198	−254	**−0.82**
310	0	310	0	0	**0**

Table 2 : Alternative routes data and plateaux raw goodness

Note that plateaux with raw goodness smaller than −0.85 (threshold) are rejected as useless alternatives. Apparently, alternatives with high sharing or high stretch are rejected as well. Once we have reduced the number of plateaux to the most interesting ones by using goodness values and admissibility criteria, we may choose to order them based on an objective function $f(\cdot)$ and on user preferences, i.e. motorways, fewer junctions, lower tolls, driving costs, familiarity. So, we introduce a new evaluation step after the plateaux identification stage and before the creation of the AG. The metrics and heuristics used for evaluation of the plateaux, of the alternative routes and the resulting alternative graph are described in detail in section 4.5.

4.4 Plateau method with penalization of edges *(hybrid approach)*

The plateau method can be combined with penalty method that was presented in chapter 3, leading to alternative routes with better characteristics.

The steps of this hybrid algorithm are still the same with plateau method with a further addition. Let $G(V, E)$ be a graph and s, t the source and destination nodes. Our goal is to find the alternative routes between s and t. So, we still compute the source and destination trees, we execute the intersection of the trees (combined tree) and we identify the plateaux .

Consider that the result of the algorithm is the alternative graph $AG(V', E')$ of G comprising the requested alternative routes (see also 3.1). Having found the plateaux chains, we should

proceed with the evaluation step based on the aforementioned criteria. Instead of sorting once the alternatives and creating the AG, the procedure evolves in rounds. In every round, we sort the alternatives based on the criteria and the objective function $f(\cdot)$, we choose the one that better satisfies them and we add it in the subgraph AG. Before moving to the next round, we penalize the edges of this alternative and we re-evaluate the alternatives. We choose again the best one and repeat the procedure until the end criterion is satisfied. The penalization's choices (presented in paragraph 3.5.1) applied in this hybrid method can vary and one should experiment in order to determine which is the best combination. We present thoroughly the implementation of the hybrid method and the chosen strategies in the next chapter.

The method of plateau with proper penalization strategy displays better results because reinforces the disjointness of the paths. Adding penalties on the edges discourages paths with many common parts to be added on the subgraph. Thus, the new method emphasizes on sharing and minimizes the amount of it among the different alternatives, while the other properties are satisfied equally well.

4.5 Criteria for alternative routes evaluation and attributes to measure in AGs.

In section 4.1, we introduced the concept of admissibility alternative routes and in section 4.3 the concept of goodness since our goal is not only to find numerous roads joining the source to the end, but compute paths that are reasonable, meaningful from a human perspective and thus, satisfy certain criteria. We want roads with low sharing, bounded stretch and high local optimality. Furthermore, alternative routes should obey the criterion of goodness that applies exclusively to the plateau and hybrid method and evaluates alternatives from another perspective. Consequently, we wish to quantify the quality both of alternative routes and alternative graphs measuring the aforementioned properties and characteristics. So, we apply metrics already defined in the previous chapter and new ones based on heuristics to achieve the best possible results.

We are going to present these attributes and criteria in the order we look at them in the practical implementation.

4.5.1 Evaluation of paths quality

In our implementation, we check each separate candidate alternative path (formed either as an extension of a candidate plateau or as a path via any node in the graph (before the construction of plateaux)) as far as the properties of stretch and goodness are concerned. For these properties, we introduce the metrics we use so as to measure them in our practical implementation.

Stretch of path.

Paths in order to be good alternatives should satisfy the property of bounded stretch. They should not be much longer than the optimal path from s to t. Thus, paths that are longer than an upper bound ($l(P) > (1 + \varepsilon) \cdot l(P_{opt})$), should be recognized as soon as possible and pruned.

We introduce this "pruning" step in the stage of the construction of the combined tree. Specifically, during this stage, the forward and backward distances for each node are added in order to compute the via node distances which are essential for the plateaux identification.

If the via node distance for a node exceeds the threshold, then this node is pruned since it cannot form a significant alternative through it. It cannot be part of a good plateau. In this way, we limit the search space and accelerate the following stages of the algorithm.

This procedure is also known as Global Thinout. Global Thinout identifies useless nodes by checking for each node v, if $v \rightarrow viaNodeDistance \leq \delta \cdot d(s,t)$ for some $\delta = (1 + \varepsilon) \geq 1$.

The next figure illustrates an example of this pruning step for $\delta = 1.2$. Red nodes are pruned since they cannot be part of a good plateau and as a result they cannot form a good alternative route from source to destination.

Figure 4.10 : The red nodes are pruned since their viaNodeDistance exceeds the threshold

$$\delta \cdot l(P_{opt}) = 1.2 \cdot 310$$

Goodness .

The goodness property is very important, since it ensures the driver that the alternative route is not found in great distance from source and destination and its main part is formed by the plateau.
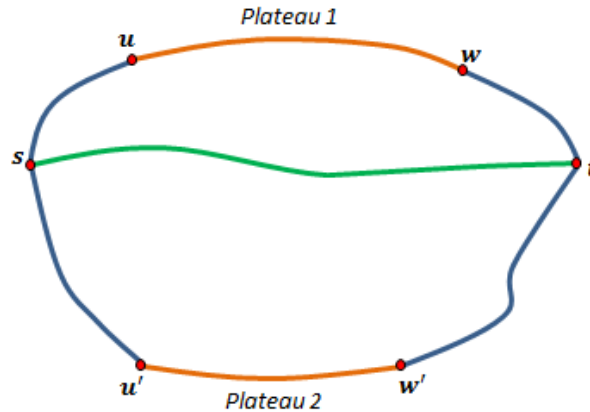
Figure 4.11 : Shortest path is illustrated by green, plateaux by orange, alternatives by the union of blue and orange. The plateau 1 has better goodness value than plateau 2.

For a plateau $u - w$ and by extension, for an alternative containing the plateau $u - w$ (as illustrated in the figure above), goodness metric is defined as

$$goodness = \frac{l_{sp} + l_{pt}}{l_{shortest\ path}}$$

l_{sp}, l_{pt} : the lengths of subpaths from source to plateau and from plateau to target respectively.

Note that in section 4.3, this ratio was called "raw goodness" and was defined with a negative sign, but for simplicity, we mention it as "goodness" and we consider it as a positive quantity.

In our implementation, we introduce the criterion of goodness in the stage of plateaux identification. That is to say, after the construction of the combined tree, we move to the creation of the plateaux chains and by extension to the creation of the corresponding candidate alternative routes. For each candidate alternative found, we compute its goodness value and if it surpasses an upper bound (threshold), we reject it as inappropriate. Obviously, in this way, the time complexity of the next evaluation step is decreased, since the search space is pruned. We have experimented with various goodness thresholds, but mainly with the values 0.85 and 1.0. The goodness criterion is further used in cases where the objective function has even values for more than one candidate alternative paths.

4.5.2 Evaluation of AG quality

Besides the evaluation of each candidate path separately, we evaluate the alternative route graph as a whole. This evaluation concerns the sharing amount and the average distance of the paths in the AG as well as the number of junctions in the AG. The metrics that are used to evaluate the AG, determine the alternatives to be added in order that the AG formed has desirable characteristics.

Sharing .

The alternative routes of the AG should have low sharing with each other and with the optimal route in order to be considered as distinct. In order to measure the amount of disjointness in the alternative graph, we present a heuristic metric referred as totalDistance :

$$totalDistance := \sum_{e=(u,v)\in E'} \frac{w(e)}{d(s,u) + w(e) + d(v,t)}$$

The total distance measures the extent to which the routes defined in the AG are nonoverlapping. Note that $d(s,u) + w(e) + d(v,t)$ is necessary because otherwise, nonoptimal paths would be encouraged. Having 2 totally disjoint paths from s to t, we have a total distance of 2. So, total distance can reach its maximum value of k when the AG consists of k disjoint paths.

In our implementation, we introduce the criterion of totalDistance at the stage of plateaux evaluation where we choose which alternatives to add in the alternative graph. Specifically, we have a set of candidate alternatives from the previous stage (plateaux identification) and in every round, we want to add the best one in the alternative graph. Thus, for each candidate alternative, we compute the new value of total distance that the current AG will have after the possible addition of the alternative in question. The alternative which yields the best value, will be added on AG. Note that totalDistance is combined with the averageDistance metric in the defined target function (both presented below) for the proper choice of the alternative to be added.

Average Distance .

Good alternatives should have bounded stretch and their lengths should be close to the optimal. Average Distance is a metric that measures the average stretch of the alternative paths in the AG. Average distance is defined using the definition of totalDistance as :

$$averageDistance := \frac{\sum_{e \in E'} w(e)}{d(s,t) \cdot totalDistance}$$

In our implementation, we introduce the criterion of $averageDistance$ at the stage of plateaux evaluation. Specifically, we have a set of candidate alternatives from the previous stage and in every round, we want to add the best one in the alternative graph, as mentioned before. Thus, for each candidate alternative, we compute the new potential value of average distance of the AG. If this new value is lower than an upper bound, and meets the other aforementioned criteria in the best way, it is added in the AG. For upper bound, we set the value of 1.1 .

Decision Edges .

Decision edges is the last metric for the AG evaluation. Decision edges measure the complexity of AG, which should be small to be digestive for a human. This metric is defined as :

$$decisionEdges := \sum_{v \in V' \setminus \{t\}} outdegree(v) - 1$$

A node in AG having more than one outgoing edges implies a decision. Human cannot handle too many of them. Thus, $decisionEdges$ set a maximum permissible number of alternatives route. In our implementation, we have set it to 10.

The next figures illustrate two examples of totalDistance, averageDistance and decisionEdges computation.
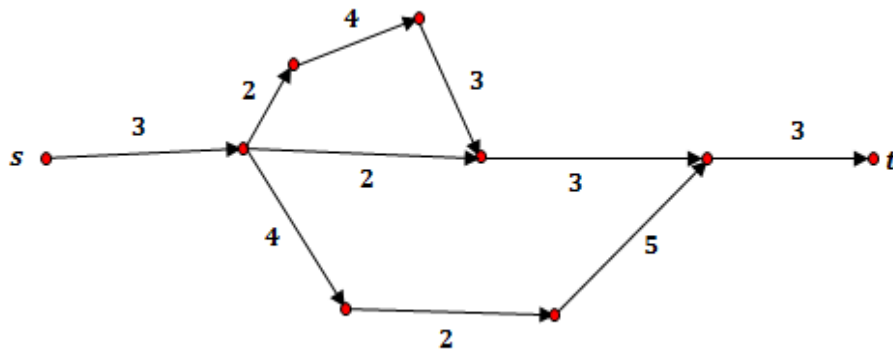
Figure 4.12 :

$$totalDistance = \frac{3+2+3+3}{3+2+3+3} + \frac{4+2+5}{3+4+2+5+3} + \frac{2+4+3}{3+2+4+3+3+3} = 1 + 0.65 + 0.5 = 2.15$$

$$averageDistance = \frac{3+2+3+3+3+4+3+4+3+5}{11 \cdot 2.15} = 1.95$$

$$decisionEdges = 2$$
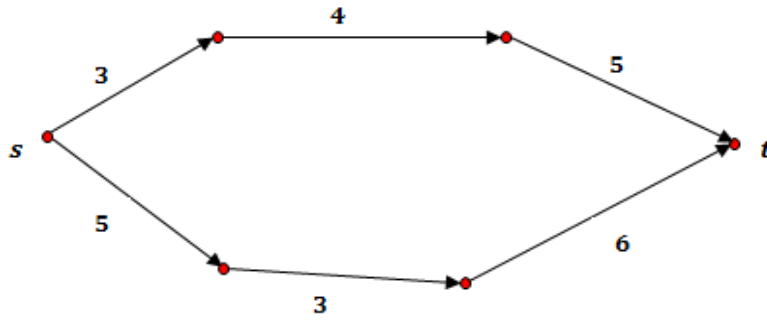
Figure 4.13 :

$$totalDistance = \frac{3 + 4 + 5}{3 + 4 + 5} + \frac{5 + 3 + 6}{5 + 3 + 6} = 2$$

$$averageDistance = \frac{3 + 4 + 5 + 5 + 3 + 6}{12 * 2} = 1.08$$

$$decisionEdges = 1$$

Objective Function .

Generally, we want alternatives with limited stretch. So, we choose a relatively small value of δ in global thinout (of course, not that small to lose good candidate alternatives) and small upper bound for averageDistance. We also want limited sharing. Thus, high values of totalDistance are more appealing. Finally, we want to limit the number of decisionEdges.

Therefore, our goal is to maximize the difference between totalDistance and averageDistance, while the other constraints are satisfied. So, we define our target function as :

$$targetFunction := totalDistance - averageDistance$$

under the constraints :

- $v \rightarrow viaNodeDistance \leq 1.2 \cdot d(s, t), for\ each\ node\ v \in plateau$
- $goodness \leq goodness\_threshold, for\ each\ alternative$
- $averageDistance \leq 1.1\ for\ AG$
- $decisionEdges \leq 10\ for\ AG$

Target function represents an overall evaluation of the AG.

In our implementation, the objective function is examined in the stage where both totalDistance and averageDistance are checked, namely in the plateaux evaluation stage. As we mentioned in the section of Total Distance, the value of targetFunction is the one that ultimately determines the alternative to be added in AG. This alternative chosen is the one which maximizes the targetFunction, while obeying the restrictions imposed.

Chapter 5

PRACTICAL IMPLEMENTATION AND METHODOLOGY

In chapter 4, we described plateau and hybrid method in theory. In this chapter, we analyze the practical implementation of these algorithms. We describe step by step all the stages of plateau algorithm from the conversion of the real road map into a graph $G(V, E)$ to the alternative graph AG that is given as a result as well as the stages of hybrid method. The programs were written in C++ and in our implementations we used a library of efficient graph structures and algorithms for large scale networks, called "pgl". This library is developed in the University of Patras[1] and already provides useful data structures for graphs (see section 2.2) and priority queues and a fast implementation of Dijkstra algorithm with many modifications and many speed-up techniques. Moreover, at a few rare places, we used STL and Boost library.

5.1 Plateau method implementation

In this section, we present the implementation strategy we followed both in plateau method and the hybrid approach. Specifically, we present the stages of the method in a more practical way with references in our practical implementations. Initially, we provide our implementation strategy for plateau method using a block diagram. In this diagram, the stages of the algorithm are presented in the order they are meet in our programs. After that, each stage is described in detail. Last but not least, we present the basic methodology for the hybrid method, namely plateau method with penalization of edges.

Figure 5.1 illustrates the stages of plateau method implemented in our C++ program.

---

[1] Pgl library is developed in the University of Patras by Prof. Zaroliagis and his research team and one can find it in the following link :   http://www.ceid.upatras.gr/faculty/zaro/software/pgl/index.html

Implementation strategy .

As stated above, the following block diagram shows the basic stages of the algorithm :

```
                    ┌─────────────────┐
                    │      Start      │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Graph Reading & │
                    │  store in data  │
                    │    structure    │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Calculation of  │
                    │ Edges weights   │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Defining source &│
                    │   target node   │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Build forward & │
                    │ backward trees  │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Build combined  │
                    │      tree       │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Find plateaux with │
                    │ goodness<=thresh │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │ Evaluate plateaux│
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │Create Alternative│
                    │   Route Graph   │
                    └─────────────────┘
                             │
                             ▼
                    ┌─────────────────┐
                    │       End       │
                    └─────────────────┘
```
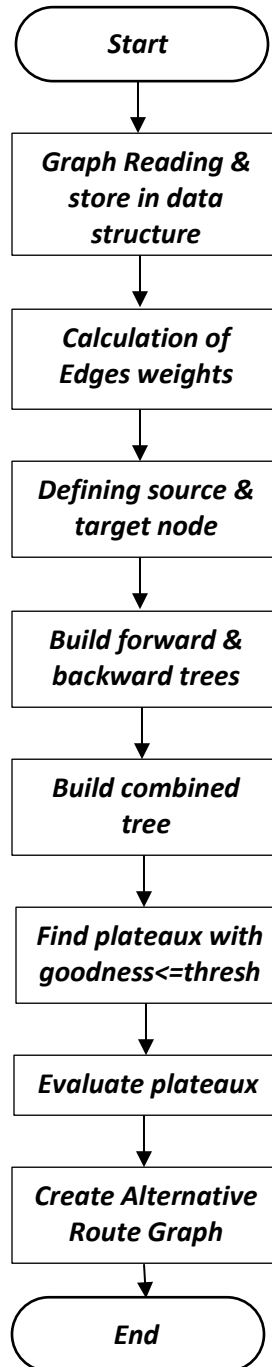
Figure 5.1 : Block diagram illustrating the implementation strategy for the plateau algorithm

Below, we are going to describe analytically the stages presented in the block diagram. Before that, we should mention a few words about the road networks given as input to the program (analytically in chapter 6). The networks are real-world street maps, such as the street map of Luxembourg, Italy, Germany etc. Each road network is given in two files. The first one (denoted by ".graph") contains $n + 1$ uncommented lines, where n is the number of nodes containing neighbor lists for each vertex from 1 to n in order. The second file (denoted by ".xyz") supply vertex coordinates in n uncommented lines. Each line i contains the coordinates x, y and z of vertex i.

Graph Reading .

As mentioned above, the graph construction is based on the two DIMACS10 files (".graph", ".xyz"). Initially, a DIMACS10 reader is created in order to access the files. This reader accesses the node lists of each vertex and the network information gathered is stored sequentially in the graph structure we choose – adjacency list, forward star, dynamic forward star or packed memory graph representation. Note that the forward star is implemented as a packed memory graph that has undergone "memory compression"[2]. The node coordinates from the .xyz file are also stored in the graph structure. Apart from the aforementioned, further information is stored in the data structure. Specifically, we define a struct for nodes and a struct for edges in order to store the required information in the data structure. The resulting graph is denoted by $G(V, E)$, $pmaG(V, E)$ (with compression), $fsG(V, E)$ or $pmaG(V, E)$ depending on the representation chosen. The numbering of nodes begins from 0 and ends at $n - 1$. This order forms the relative position of the nodes.

The struct for nodes("struct Node") contains the following fields :

- unsigned int dist, distBack, viaNodeDistance;
  where   dist : minimum distance of each node from source node
          distBack : minimum distance of each node from target node
          viaNodeDistance : sum of dist and distBack (dist+distBack)

- void* pred, succ;
  where   pred, succ : pointers of each node to other nodes, i.e. if Dijkstra executed
          pred : pointer to predecessor in a forward shortest path tree
          succ : pointer to successor in a backward shortest path tree

- void* startNode, link;
  where   startNode, link : pointers to nodes. Useful in order that each node belonging in
          a plateau store the start and the end of the corresponding plateau

---

[2] The graph data structures that we used in our implementations (adjacency list, forward star, dynamic forward star and graph memory representation), were borrowed from the "pgl library".

- unsigned int plateauFlag;

  where   plateauFlag : flag for search and check of the nodes belonging in a plateau

- unsigned int selectionID;

  where selectionID : integer, useful for the class NodeSelection used in the implementation

The struct for edges ("struct Edge") contains one field :

- unsigned int weight;

  where   weight : the weight of the edge

Calculation of Edges weights.

The method "calcWeights" is responsible for the calculation of edges' weights. The calculation is based on the Euclidean distance between the start and the end node of each edge. Specifically, let (u, v) be an edge of the graph. The weight of (u, v), namely $w(u, v)$, is equal to $euclideanDistance(u{-}{>}x, u{-}{>}y, v{-}{>}x, v{-}{>}y)$ where x, y are the coordinates of each node.

Defining source & target node .

The program execution and the computation of alternative routes make sense between two nodes of the graph which correspond to the start and the destination point respectively. So, it is necessary we have a source and a target node. These two nodes are chosen randomly using a generator of random numbers. This generator returns 2 values of type double between 0 and 1, the first corresponding to the source node and the latter to the target node. These random values are then multiplied by the number of nodes n so as to generate the relative positions of source and target node in the graph representation.

In case of multiple, consecutive $s - t$ queries, the random numbers of the generator are stored initially in a vector "queries" and in every iteration of the algorithm, the appropriate pair of numbers is retrieved and the corresponding pair of source and target nodes is computed for the execution of the method.

Build Forward & Backward trees .

The first two basic steps of plateau algorithm include the construction of the shortest path tree from source node to target node and the computation of the shortest path trees from all the other nodes to the target node, as described analytically in chapter 4.

As far as the forward tree is concerned, an execution of Dijkstra algorithm is necessary until target node is settled. For this computation, we use a C++ class, called Dijkstra[3] and specifically the method $runQuery(source, target)$ which develops a shortest path tree from the source to the target (forward tree). For each node that becomes settled in the forward tree, the field "dist" obtains a new value equal to the shortest path distance of the node from the source. Moreover, the field "pred" of each node points to its predecessor in the forward path tree, according to Dijkstra execution.

Similarly, for the construction of the shortest path trees from all the other nodes to the target node, Dijkstra execution is needed. Since the graph is undirected, it is equivalent to run a Dijkstra from the target node (to all the others) without any modification to the direction of the edges. For this computation, we use a C++ class, called BackwardDijkstra and specifically its method runQuery which stops the Dijkstra execution when source node is settled. After the end of execution, the backward shortest path tree is built and the field "distBack" of each node is equal to its shortest distance from target node. The field "succ" of each node also points to its successor node (predecessor in the backward tree from another point of view).

For the next stages of the plateau algorithm, a new class has been created. It is called "Plateau" and implements the appropriate methods for the combined tree construction and the plateau identification and evaluation.

Build Combined Tree .

First of all, we remind that plateau is a path where the sum of dist and distBack, namely the viaNodeDistance, of its nodes stays stable along it. Note that the via node distance of a node v expresses the distance a driver traverses from source to target via/through this node v So, in order to identify the plateau chains, it is necessary we compute and store the via node distances. Specifically, we implemented a method denoted by "buildCombinedTree" in "Plateau" class, where all the nodes of the graph are checked. In more detail, for the nodes which are settled both from the forward and the backward Dijkstra, we sum up the dist and distBack values from the corresponding fields and if the resulting via node distance is smaller than an upper bound (limited stretch, see also 5.3.1), then we store it in the corresponding struct field. In other words, we apply a first level of node pruning since we reject nodes that form single via paths and by extension, plateaux which length is much longer than the length of the optimum/shortest path. At the same time, the nodes with admissible via node distances, are inserted in a priority queue pq according to the minimum distance from the source node.

---

[3] The classes Dijkstra and BackwardDijkstra were also borrowed from the "pgl library".

Find Plateaux .

From the previous stage, the priority queue pq, contains all the nodes that potentially belong in a plateau. In every round (until the priority queue is empty), the node u with the minimum distance from the source node is extracted from the priority queue and by following the "succ" pointers, we check its successors – as defined in the backward shortest path tree (follow the shortest path from the current node u to target). If the successors have the same via node distance with node u, a new plateau is created. It is sufficient to find only one additional node with the same via node distance in order to create a new plateau (at least u and $u->succ$). At the same time, the pointers "startNode" and "link" of the nodes that belong in the currently formed plateau, acquire their new values. The pointers "startNode" and "link" of each node point at the start and the end of the plateau to which it belongs (specifically, for the intermediate nodes of a plateau, only the pointer "link" is used so as to point at the start of the plateau they belong to). Furthermore, all the nodes belonging in plateaux set their flag "plateauFlag" to 1 and are then removed from the priority queue so as not to be checked twice. If a successor has different via node distance from u or it has the same via node distance but belongs to a different plateau, the procedure continues with a new extraction from the priority queue pq. A special case that we should take into account so as to ensure correct results is the following. Consider a node u that has two equivalent successors (same via node distance), choses to follow (successor) one of them by random (or according to which node the pointer "succ" was set to point to in the backward tree) and as a result it is likely that another plateau with the same via node distance exists (probably with different goodness). In order to avoid this situation, apart from the node's successors, we also check only the predecessor of the initial node u. In case the two nodes, u and $u->pred$, have the same viaNodeDistance, further checks take place in order to identify the maximum plateau that can be constructed.

The plateaux identified may be numerous. For this reason, we apply a second level of pruning. The plateaux that are not admissible are rejected. In this stage, the admissibility of a plateau is defined by its goodness value. Thus, if the goodness value of the candidate plateau is bigger than an upper bound, the plateau is rejected and is not further evaluated in order to be added in the alternative route graph. The start nodes of the admissible plateaux ($goodness <= upper\ bound$) are stored in a vector instantiated as a variable of the NodeSelection class (this class is created in "pgl library" so as to store a smaller set/selection of nodes of the initial graph and implements various methods, such as selectNodes, getMembers, isMember etc.). This selection of vertices is provided to the next stage of the algorithm which is responsible for the final evaluation of the candidate plateaux.

Evaluate Plateaux .

In this stage, we evaluate the alternative route graph that results from the addition of a new alternative route. The alternative routes are formed by the union of the corresponding

candidate plateaux with the source and target nodes. This evaluation is based on the "totalDistance", "averageDistance" and "decisionEdges" criteria, described in detail in section 4.5. In particular, in every round, our aim is to add that alternative route, or for simplicity, that plateau) that maximizes the target function : $targetFunction = totalDistance - - averageDistance$.

The criteria are initialized to zero. Then, the nodes contained in the NodeSelection variable created in the previous stage, are checked in turn. The process is the following : it starts from the in question node u and sums up the weights of the edges initially in the direction of its predecessors ($u -> pred$) and then in the direction of its successors ($u -> succ$) and stores the result in a variable denoted by "weightSum". Note that the computation of weight sum in each direction stops when a node (predecessor in the first case, successor in the latter) was formerly checked, that is to say, the weights of the edges from this predecessor (or successor) until the source (or the target) have already been included in totalDistance and averageDistance. Having calculated weightSum, we can compute the new values of totalDistance and averageDistance :

- $totalDistanceNew = totalDistance + \frac{weightSum}{u \rightarrow viaNodeDistance}$

  Where $totalDistance$ refers to the value of the $totalDistance$ metric of the formed so far alternative route graph

- $averageDistanceNew = \frac{weightSum + AG\_weightSum}{totalDistanceNew * shortest\_path}$

  Where $AG\_weightSum$ refers to the sum of edges' weights of the alternative route graph formed so far. Note that the numerator of the $averageDistanceNew$ metric of the chosen alternative in every round is stored in a variable so as to be used in the next round.

The plateau and by extension the alternative route that achieves the best value for the targetFunction with respect to the constraints imposed for averageDistance and decisionEdges is chosen for the alternative route graph and removed from the NodeSelection variable.. In this stage, we choose the alternative route by setting the plateauFlag field of its nodes to 2. Obviously, the shortest path (one of them in case there are more than one) is the first to be chosen. Note that for every alternative route added (except the first one which is the shortest path) the value of decisionEdges increase by one. The process ends when the value of decisionEdges exceeds the upper bound imposed or if no other candidate alternative satisfy the criteria and the constraints. As mentioned above, the nodes that belong to the admissible alternatives have "plateauFlag" set to 2.

Create Alternative Route Graph .

In this stage of the implementation, we create a subgraph $subG$ of the initial graph G (or fsG, $pmaG$ etc.) consisting of the nodes and the edges of the admissible alternative routes identified in the previous step. This subgraph is implemented as an adjacency list with appropriate node and edge structs. The node struct has only one field, a pointer "nodeDesc" to its corresponding node in the initial graph. The edge struct is the same as in the initial graph and has only one field where the weight of each edge is stored.

For the creation of the alternative route graph, we implemented a new class, called Subgraph. In this class, we implemented the three necessary steps for the alternative route graph construction.

Node Insertion . The initial graph is traversed and nodes with plateauFlag value equal to 2, are inserted in the subG. A pointer to the corresponding node in the initial graph is also stored in the field "nodeDesc". Moreover, the mappings between the nodes of the initial graph and the subgraph are stored in a vector.

Sort . Then, the vector with the mappings is sorted with the use of the library function "sort" in ascending order of the nodes descriptors in the initial graph.

Edge Insertion . The aforementioned vector is traversed. For each node, we check if its edges should exist in the alternative graph. Actually, we check if the end node of the edge (the start node is the current node of the vector) has plateauFlag value equal to 2. If yes, we add this edge on the alternative route graph with the same weight.

5.2 Plateau method with penalization of edges *(hybrid method)*

In this hybrid method, as mentioned before, we want to amplify the disjointness of the alternative routes and in order to achieve it we penalize some edges according to a strategy. Practically, in order to combine plateau method with penalty in our C++ implementation, we should, first of all, add one more field in the "struct edge" of the input graph. This field, denoted by penalty, is an unsigned integer for storing the edge weight after the penalization.

Generally, the method's stages are common to these of plateau method, as illustrated in figure 5.1, except the one doing the evaluation of the alternative routes. As far as this stage is concerned, for each alternative we choose to add in the subgraph, we traverse its edges and multiply their weights by a penalty factor (equal to 0.3). Moreover, we multiply the outgoing and incoming edges of the nodes belonging to the alternative route by a smaller factor called rejoin factor ($0.005 * penalty\ factor$) so as to prevent alternatives "direct towards" an alternative already belonging in the AG. Furthermore, for the containing nodes of the alternative route, we set plateauFlag to 2, as in plateau method implementation. So, the question is how to exploit these penalized weights in selecting the alternative (since alternatives may consist of

common parts) and how they will be adapted to the criteria used (totalDistance, averageDistance, decisionEdges). We apply two alternative approaches:

- 1st approach : we modify the calculation and the meaning of totalDistanceNew and hence the objective function for selecting paths. Specifically, we use the penalized weights for the calculation of totalDistanceNew. So :

$$penalizedTotalDistanceNew = totalDistance + \frac{weightSum}{penalizedPathLength}$$

Where $totalDistance$ refers to the metric's value on the formed so far alternative route graph based on the normal weights and distances.

Therefore ,
$$penalizedTargetFunction = penalizedTotalDistance - averageDistance$$

In this way, the introduction of more disjoint alternative routes is promoted while at the same time the constraints are satisfied. In this first approach, the selection of an alternative is also accompanied by penalization of its edges (+ rejoin penalty).

- 2nd approach : we change the objective function for the final selection of alternatives and instead of maximizing the aforementioned $targetFunction = totalDistance - averageDistance$, we defined a new $targetFunction = penalizedPathLength$, where $penalizedPathLength$ is the length of each alternative route after penalization. Specifically, having created the NodeSelection variable from the previous stage, in each round, we check all the remaining candidate alternatives which arise from the corresponding plateaux. For each alternative, we calculate its length (from the source to the target), which may include common edges with an alternative already been added to the alternative route graph. These common edges are penalized, so the path length is also penalized. Of all these alternatives, we look for the one that minimizes the $targetFunction = penalizedPathLength$ while respecting the constraints for $averageDistance$ and $decisionEdges$ ($averageDistance$ calculation is still based on the original graph weights). The alternative selected is punished as mentioned above with an appropriate penalty (+ rejoin penalty).

Chapter 6

EXPERIMENTS AND RESULTS

In this chapter we give an experimental evaluation of our programs in different road networks. Specifically, we evaluate the alternative graph and paths quality based on the metrics presented in section 4.5 for different goodness thresholds as well as the time performance and memory usage resulting from the execution for the different graph data structures implemented. Initially, we describe the experimental setup, including the experimental environment, input data and other necessary parameters for the experiments. Then, we present tables with our experimental results and we make comparison of them, leading to useful conclusions. The section is wrapped up by a summary on the main results.

6.1 Experimental setup

6.1.1 Experimental environment

Experiments have been done on a quad-core Intel® Xeon® processor X3363 clocked at 2.83GHz with 8GB RAM and 6MB Cache, running "Ubuntu precise (12.04.2 LTS)". All of our programs are single threaded and thus, only one of the cores was used. The program was compiled by the GNU C++ compiler 4.6.3 using optimization level 3.

6.1.2 Input

In this section we introduce the input data we use throughout our experiments. All networks are based on real world data and are, thus, not synthetic. The source of these networks is the Center of Discrete Mathematics & Theoretical Computer Science (DIMACS – http://www.cc.gatech.edu/dimacs10/archive/streets.shtml).

The networks are real-world street maps, such as the street map of Luxembourg, Italy, Germany etc. These maps are undirected and unweighted versions of the largest strongly connected component of the corresponding Open Street Map road networks (http://download.geofabrik.de).

The road network provided is given in two files following the popular METIS input/output format. The first one (denoted by ".graph") contains $n + 1$ uncommented lines, where n is the number of nodes. The first of these lines contains two integers, separated by space that denote the number of vertices and the number of edges in the graph. Note that in this case the number of edges is only the half of the sum of the vertex degrees (since the graph is undirected and edges are stated twice from the two ends of each edge). The remaining n lines contain neighbor lists for each vertex from 1 to n in order. These lists are sets of integers separated by spaces and contain all the neighbors of a given vertex. The second file (denoted by ".xyz") supply vertex coordinates in n uncommented lines. Each line i contains the coordinates x, y and z of vertex i. In the street maps used, the z coordinate is always set to 0 (altitude is not taken under consideration).

Table 3 contains the size of our test instances.

Map	No. of Nodes	No. of Edges
Luxembourg	114599	119666
Belgium	1441295	1549970
The Netherlands	2216688	2441238
Italy	6686493	7013978
Germany	11548845	12369181

Table 3 : Size of our main input graphs

6.1.3 Queries

The results, either concerning the quality of the alternative route graph and its paths or the time performance and memory usage, are computed by running a number of random queries. For the "small" graphs, such as Luxembourg and Belgium, we use 1000 random queries, but on the "larger" graphs we only run 100 random queries. Note that the $s - t$ pairs, are chosen randomly at the beginning. So, we have 5 different sets of random s-t pairs (each set consists of 1000 or 100 pairs respectively), one for each of the 5 road networks in order to be able to compare the results on a specific map. That is to say, we run the same queries for all the executions on a specific road network.

6.1.4 Measurements and statistics

As mentioned above, in our experimental results, we report two aspects. The first one refers to the quality of the resulting AG and the latter to the execution times and the memory usage.

As far as AG and paths quality is concerned, for every execution of the 1000 (or 100) queries, we measure:

- the goodness values for all the plateaux detected
- the stretch for all the alternative routes detected
- the targetFunction value for every resulting AG
- the totalDistance value for every resulting AG
- the averageDistance value for every resulting AG
- the decisionEdges value for every resulting AG

Based on these measurements, we export statistics for the various implementations (plateau, plateau with penalization of edges (1st approach), plateau with penalization of edges (2nd approach)) and compare them. Specifically, we detect the minimum and maximum value for each of the 5 metrics and we compute the average value of the goodness, stretch, targetFunction, averageDistance, decisionEdges metrics and their variance, where needed. Note that the value of (average) totalDistance and by extension, the (average) value of targetFunction should be examined together with the (average) value of decisionEdges. That is to say, we are interested not only in the percentage of disjointness found in the AG (illustrated by totalDistance metric), but also in the number of the alternatives achieving this percentage. For example, if totalDistance is equal to 4 and alternatives(or equivalently decsionEdges) are equal to 4, it means that we have 4 totally disjoint alternative paths. On the contrary, if totalDistance is equal to 4 and decisionEdges are equal to 8, it means that the alternative routes are common almost by 50%.

Furthermore we compute the time performance of the programs depending on the graph data structure used. We execute queries on the road maps for the four graph representations – adjacency list, forward star, dynamic forward star and packed-memory graph (see section 2.2), and we measure the time needed for each of the algorithm's stages (forward, backward tree, combined tree, plateaux finding, evaluation etc.) as well as the total time performance. Moreover, we examine the memory needed for the graph representation depending on the data structure used. For these measurements, we export the corresponding statistical data, we make comparisons and draw conclusions.

6.2 Experiments

We conducted experiments so as to measure the aforementioned criteria and to export statistical data, necessary for evaluation of the methods. For our experiments, we executed queries with specific upper bounds for the admissibility criteria. Specifically, we require :

- $\delta = 1.2$, referring to the upper bound for alternative path's stretch
- $averageDistance \leq 1.1$
- $decisionEdges \leq 10$
- Minimization or maximization of the objective function :
  - In case of Plateau method, maximization of the following objective function is required : $targetFunction = totalDistance - averageDistance$

- In case of Plateau method with penalization of edges (1st approach), maximization of the following objective function is required:
  $$penalizedTargetFunction = penalizedTotalDistance - averageDistance$$
- In case of Plateau method with penalization of edges (2nd approach), minimization of the following objective function is required :
  $$targetFunction = penalizedPathLength$$

As far as the goodness criterion is concerned, we conducted experiments with two different upper bounds, equal to 1.0 and to 0.85 . The goodness criterion refers to this amount of the alternative path that does not belong to the plateau chain. So, the lower the upper bound is set, the smaller the lengths of the subpaths from source to plateau and from plateau to target are allowed to be.

6.2.1 AG and alternative paths quality *(goodness_threshold = 1.0)*

Below we present tables containing results for the AG and alternative paths quality for the various implementations.

Plateau method *(goodness_threshold = 1.0)* **.**

The following two tables show the statistical data referring to AG and alternative paths quality resulting from the execution of plateau method in 5 different road networks for $goodness\_threshold = 1.0$

	Alternative Route Graph Quality									
Map	**TargetFunction**				**AverageDistance**		**DecisionEdges**			
	min	*max*	*average*	*variance*	*Average*	*variance*	*min*	*max*	*average*	*variance*
Luxembourg	0.01	5.69	3.38	1.13	1.07	0.00039	1	10	9.64	2.20
Belgium	0.00	7.35	5.43	0.86	1.06	0.00028	0	10	9.98	0.04
The Netherlands	0.73	7.06	4.65	0.93	1.06	0.00317	8	10	10.00	0.00
Italy	1.62	6.41	4.50	0.94	1.06	0.00031	10	10	10.00	0.00
Germany	3.09	7.04	5.35	0.59	1.06	0.00026	10	10	10.00	0.00

Table 4 : Statistics on the quality of the alternative route graph resulting from the execution of plateau method on various European road networks

Alternative Routes Quality			
Map	**Goodness**		**Stretch**
	average	*variance*	*average*
Luxembourg	0.82	0.085	1.07
Belgium	0.85	0.082	1.06
The Netherlands	0.85	0.083	1.06
Italy	0.86	0.080	1.06
Germany	0.86	0.081	1.06

Table 5 : Statistics on the quality of the alternative routes resulting
from the execution of plateau method on various European road networks

To be more specific, each line indicates the results computed from the execution of 1000 queries in the maps of Luxembourg and Belgium and 100 queries in the maps of Netherlands, Italy and Germany. As shown, these results contain minimum and maximum values as well as the average and the variance of the different metrics.

Remarks .

- The average goodness values are almost at the same level for all the maps.
- Mean averageDistance is the same as mean stretch for each map, since they measure the same property each one from another perspective.
- As the size of the map increases, the mean value of decisionEdges also increases. This is reasonable since in larger maps, the probability of finding more alternative routes is higher.

Plateau method with penalization of edges *(1ˢᵗ approach – goodness_threshold = 1.0)* **.**

The following two tables show the statistical data referring to AG and alternative paths quality resulting from the execution of the 1ˢᵗ approach of the hybrid algorithm (see sections 4.4, 5.5) in 5 different road networks for $goodness\_threshold = 1.0$

Alternative Route Graph Quality										
Map	**TargetFunction**				**AverageDistance**		**DecisionEdges**			
	min	*max*	*average*	*variance*	*average*	*variance*	*min*	*max*	*average*	*variance*
Luxembourg	0.01	6.12	3.58	1.44	1.07	0.00046	1	10	9.64	2.19
Belgium	0.00	7.74	5.74	1.11	1.06	0.00029	0	10	9.98	0.14
The Netherlands	0.73	7.01	4.78	1.50	1.06	0.00032	8	10	9.98	0.04
Italy	1.79	6.60	4.67	1.18	1.06	0.00033	10	10	10.00	0.00
Germany	3.41	7.36	5.67	0.74	1.06	0.00029	10	10	10.00	0.00

Table 6 : Statistics on the quality of the alternative route graph resulting from the execution of
the 1ˢᵗ approach of the hybrid method on various European road networks

	Alternative Routes Quality		
Map	**Goodness**		**Stretch**
	average	*Variance*	*average*
Luxembourg	0.82	0.084	1.08
Belgium	0.85	0.082	1.06
The Netherlands	0.86	0.082	1.06
Italy	0.86	0.080	1.06
Germany	0.86	0.081	1.06

Table 7 : Statistics on the quality of the alternative routes resulting from the execution of the 1st approach of the hybrid method on various European road networks

Remarks .

Making a comparison between the plateau method and the 1st hybrid method, we conclude that for the same configuration :

- The average value of targetFunction is better by about 5% in the hybrid method for all the road networks. The difference between the values of targetFunction in the two implementations derives mainly from the better mean totalDistance values in the hybrid method. This fact implies that the hybrid method actually manages to select more disjoint alternative routes than the plateau method, achieving better values for totalDistance and as a result, targetFunction.
- As far as the other metrics are concerned, namely averageDistance, decisionEdges, goodness and UBS remain almost the same for both methods.

Plateau method with penalization of edges *(2nd approach – goodness_threshold = 1.0)* **.**

The following two tables show the statistical data regarding AG and alternative paths quality resulting from the execution of the 2nd hybrid approach (see sections 4.4, 5.5) in 5 different road networks for $goodness\_threshold = 1.0$. Note that the 2nd approach uses a totally different objective function which selects alternative routes with the minimum length while satisfying the other criteria.

	Alternative Route Graph Quality									
Map	**TargetFunction**				**AverageDistance**		**DecisionEdges**			
	min	*max*	*average*	*variance*	*average*	*variance*	*min*	*max*	*average*	*variance*
Luxembourg	0.01	5.42	3.26	1.08	1.07	0.00040	1	10	9.64	2.20
Belgium	0.00	7.43	5.19	1.07	1.05	0.00030	0	10	9.98	0.14
The Netherlands	0.76	6.36	4.14	1.31	1.05	0.00030	8	10	9.98	0.04
Italy	1.56	6.10	4.03	1.11	1.05	0.00035	10	10	10.00	0.00
Germany	2.66	6.95	5.12	0.87	1.05	0.00028	10	10	10.00	0.00

Table 8 : Statistics on the quality of the alternative route graph resulting from the execution of the 2nd approach of the hybrid method on various European road networks.

Alternative Routes Quality			
Map	**Goodness**		**Stretch**
	Average	*Variance*	*average*
Luxembourg	0.79	0.085	1.07
Belgium	0.82	0.083	1.06
The Netherlands	0.83	0.082	1.05
Italy	0.83	0.082	1.05
Germany	0.84	0.082	1.05

Table 9 : Statistics on the quality of the alternative routes resulting from the execution of the 2nd hybrid method on various European road networks

Remarks .

In comparison with plateau method and the 1st hybrid approach, the 2nd hybrid approach :

- has slighter better mean averageDistance and goodness values. That is to say, it selects a bit shorter alternatives which at the same time have shorter lengths from source to plateau and from plateau to target.
- has worse mean targetFunction values by about 6% than plateau method. In other words, the penalization of edges in combination with the in question objective function does not improve the quality of the resulting AG.

Conclusion .

Based on the experimental results for the AG quality, we conclude that the three approaches are almost equally good. However, out of the three, the 1st approach of plateau with penalization of edges (+ rejoin penalty) yields the best results as far as the disjointness of the paths is concerned (which is the most meaningful metric, since the others are limited by upper bounds). In the next section, we will examine how the goodness threshold influences the quality of the resulting graph.

6.2.2 AG and alternative paths quality *(goodness_threshold = 0.85)*

In this section, we present results concerning the AG and alternative routes quality, conducting experiments in the 5 road networks for a smaller goodness_threshold (=0.85). Our goal is to examine how the reduction of the goodness upper bound influences the selection of alternatives and as a result the measured quality, as reflected by mean targetFunction and mean decisionEdges. Obviously, we will not focus on comparing which of the three methods is better for goodness_threshold=0.85, since, the 1st approach of hybrid method is expected to be slightly better, based on the results for goodness_threshold=1.0. Instead, we will compare the results

with the corresponding results for goodness_threshold = 1.0 for the three implementations.

Plateau method *(goodness_threshold = 0.85)* **.**

The following two tables show the statistical data referring to AG and alternative paths quality resulting from the execution of plateau method in 5 different road networks for $goodness\_threshold = 0.85$

Map	Alternative Route Graph Quality									
	TargetFunction				AverageDistance		DecisionEdges			
	min	*max*	*average*	*variance*	*Average*	*variance*	*min*	*max*	*average*	*variance*
Luxembourg	0.00	4.67	1.80	0.77	1.04	0.0005	0	10	5.03	6.52
Belgium	0.00	5.75	3.15	1.07	1.03	0.0003	0	10	8.95	4.09
The Netherlands	0.21	5.48	2.53	1.67	1.03	0.0003	1	10	8.74	3.94
Italy	0.00	5.33	2.37	1.44	1.03	0.0004	2	10	8.59	4.42
Germany	0.73	5.69	2.88	1.14	1.02	0.0003	3	10	9.40	2.02

Table 10 : Statistics on the quality of the alternative route graph resulting from the execution of plateau method on various European road networks

Map	Alternative Routes Quality		
	Goodness		Stretch
	average	*variance*	*average*
Luxembourg	0.57	0.097	1.04
Belgium	0.61	0.087	1.02
The Netherlands	0.56	0.108	1.02
Italy	0.53	0.112	1.03
Germany	0.58	0.100	1.02

Table 11 : Statistics on the quality of the alternative routes resulting from the execution of plateau method on various European road networks

Plateau method with penalization of edges *(1ˢᵗ approach – goodness_threshold = 0.85)* **.**

The following two tables show the statistical data referring to AG and alternative paths quality resulting from the execution of the 1ˢᵗ approach of the hybrid algorithm (see sections 4.5, 5.5) in 5 different road networks for $goodness\_threshold = 0.85$

	Alternative Route Graph Quality									
Map	**TargetFunction**				**AverageDistance**		**DecisionEdges**			
	min	*max*	*average*	*variance*	*Average*	*variance*	*min*	*max*	*average*	*variance*
Luxembourg	0.00	5.31	1.95	0.92	1.04	0.0005	0	10	5.02	6.52
Belgium	0.00	5.95	3.40	1.22	1.03	0.0003	0	10	8.95	4.09
The Netherlands	0.21	5.62	2.67	1.86	1.03	0.0004	1	10	8.75	4.51
Italy	0.00	5.34	2.53	1.59	1.03	0.0004	2	10	8.59	4.42
Germany	0.73	5.86	3.02	1.25	1.02	0.0003	3	10	9.40	2.02

Table 12 : Statistics on the quality of the alternative route graph resulting from the execution of the 1ˢᵗ hybrid approach on various European road networks

	Alternative Routes Quality		
Map	**Goodness**		**Stretch**
	Average	*variance*	*average*
Luxembourg	0.57	0.243	1.04
Belgium	0.62	0.082	1.02
The Netherlands	0.57	0.105	1.02
Italy	0.54	0.110	1.03
Germany	0.54	0.099	1.03

Table 13 : Statistics on the quality of the alternative routes resulting from the execution of the 1ˢᵗ hybrid approach on various European road networks

Plateau method with penalization of edges *(2ⁿᵈ approach – goodness_threshold = 0.85)* **.**

The following two tables show the statistical data referring to AG and alternative paths quality resulting from the execution of the 2ⁿᵈ approach of the hybrid algorithm (see sections 4.5, 5.5) in 5 different road networks for $goodness\_threshold = 0.85$

Alternative Route Graph Quality									
Map	**TargetFunction**				**AverageDistance**		**DecisionEdges**		
	min	*max*	*average*	*variance*	*Average*	*variance*	*min*	*max*	*average*	*variance*
Luxembourg	0.00	5.32	1.95	0.92	1.04	0.0005	0	10	5.02	6.52
Belgium	0.00	5.79	3.27	1.01	1.03	0.0002	0	10	8.95	4.09
The Netherlands	0.21	5.13	2.55	1.54	1.03	0.0003	1	10	8.75	4.51
Italy	0.00	4.92	2.43	1.32	1.03	0.0004	2	10	8.59	4.42
Germany	0.73	5.23	2.90	1.02	1.02	0.0003	3	10	9.40	2.02

Table 14 : Statistics on the quality of the alternative route graph resulting from the execution of the 2nd hybrid approach on various European road networks

	Alternative Routes Quality		
Map	**Goodness**		**Stretch**
	Average	*variance*	*average*
Luxembourg	0.57	0.240	1.04
Belgium	0.62	0.252	1.03
The Netherlands	0.58	0.107	1.03
Italy	0.54	0.104	1.03
Germany	0.60	0.089	1.02

Table 15 : Statistics on the quality of the alternative routes resulting from the execution of the 2nd hybrid approach on various European road networks

Remarks .

- We notice that the average values of decisionEdges are smaller than the corresponding for goodness_threshold=1.0. By appropriate calculations between the respective tables , we estimate that mean decisionEdges for the new goodness threshold are reduced by about 50% in the road network of Luxembourg for all of the three methods, while in the other maps, the percentage fluctuates around 10%. Given that the map of Luxembourg is quite small, the majority of alternative routes consist of relatively small plateaux. As a result, when the upper bound is set to 1.0, the algorithm identifies paths that satisfy the criterion, but when this bound is set to a lower value, like 0.85, a lot of them are not still admissible. On the other hand, in cases of maps with greater number of nodes and edges, the difference between the two executions is much smaller.
- The new mean averageDistance values differ slightly compared with those obtained for goodness_threshold = 1.0.
- Average values of targetFunction are smaller than the corresponding for goodness_threshold = 1.0. By appropriate calculations, we estimate that mean

targetFunction values are reduced by about 45%-50% for all the maps and for all the methods. Given the reduction in decisionEdges, we conclude that in Luxembourg a constant ratio between targetFunction to decisionEdges is maintained and the results are equivalent in the two cases. On the contrary, in the other maps, the decrease in targetFunction is not proportional to the decrease of decisionEdges and thus the new resulting AG deteriorates in quality as regards the disjointness. Specifically as the size of the maps increases, alternative routes satisfying the new more strict threshold can be found (desicionEdges value stays almost the same), but these alternatives are more overlapping, leading to worse quality.

Conclusion .

Generally, there is no obvious policy that brings the best possible results. On the contrary, it depends on the specifications that each alternative route graph must fulfill. Apparently, decreasing the goodness upper bound, alternative routes consist of larger plateaux, but may have common parts with each other, as demonstrated by the experiments in these maps. On the other hand, setting a more relaxed threshold, alternatives may possibly consist of smaller plateaux, but may be better on the other criteria.

6.2.3 Memory usage and time performance of plateau method *(goodness_threshold = 1.0)*

In the following section, we present the experimental results concerning the time performance and the memory usage depending on the data structure used for the graph representation. Initially, we record the memory needed from each data structure to store the graph. Besides this, we aim to find out the time improvement or deterioration depending on the graph representation. Thus, one method is sufficient so as to induct conclusions concerning time performance and memory usage. So, we have conducted experiments using plateau method. The four data structures used are adjacency list, forward star, dynamic forward star, packed-memory graph. The goodness threshold is set to 1.0.

6.2.3.1 Memory usage

The memory needed for each graph representation is presented below :

	Representation	Memory Usage*(Mbytes)*
	Adjacency List	24.07
Luxembourg	**Forward Star**	24.07
	Dynamic Forward Star	25.82
	Packed-Memory Graph	27.00

Table 16 : Luxembourg memory usage

	Representation	Memory Usage*(Mbytes)*
	Adjacency List	306.85
Belgium	**Forward Star**	306.85
	Dynamic Forward Star	328.84
	Packed-Memory Graph	432.00

Table 17 : Belgium memory usage

	Representation	Memory Usage*(Mbytes)*
	Adjacency List	477.18
The Netherlands	**Forward Star**	477.18
	Dynamic Forward Star	511.00
	Packed-Memory Graph	864.00

Table 18 : The Netherlands memory usage

	Representation	Memory Usage*(Mbytes)*
	Adjacency List	1407.36
Italy	**Forward Star**	1407.36
	Dynamic Forward Star	1509.39
	Packed-Memory Graph	1728.00

Table 19 : Italy memory usage

	Representation	Memory Usage*(Mbytes)*
	Adjacency List	2454.09
Germany	**Forward Star**	2454.09
	Dynamic Forward Star	2630.31
	Packed-Memory Graph	3456.00

Table 20 : Germany memory usage

Remarks .

- For all the road networks, the theoretical analysis is confirmed. The adjacency list and the forward star representation need the minimum amount of memory to be stored, while the packed-memory graphs needs the largest amount of memory to be stored.

This is rational, since the packed-memory graph binds some empty slots of memory (holes) in order to achieve efficient updates on the graph. Dynamic forward star also binds empty slots and as a results uses more memory than the static forward star and the adjacency list

- Although adjacency list and static forward star need exactly the same amount of memory to store the graph, the layout of the graph in memory in the adjacency list representation is random while in the forward star representation is as compact as possible.

- In our implementation, forward star is implemented as a compressed packed-memory graph. That is to say, the empty cells are not uniformly distributed, but they are at the end of the data structure and as a result forward star is compact. For that reason, we consider that forward star uses exactly the same amount of memory as the adjacency list.

6.2.3.2 Time performance *(goodness_threshold = 1.0)*

The execution times refer to each of the stages of the plateau method, namely, construction of forward, backward tree and combined tree, plateaux finding and evaluation, construction of AG. The total execution time is also included in the following tables. Time is given in sec.

Luxembourg .

Time Performance (sec)							
Graph Representation	Forward & Backward Tree	Combined Tree	Plateaux Finding	Plateaux evaluation	AG construction	Total time	
	average	average	average	average	average	average	variance
Adjacency List	0.063	0.008	0.007	0.001	0.009	0.088	0.002
Forward Star	0.041	0.006	0.008	0.002	0.005	0.061	0.001
Dynamic Forward Star	0.055	0.008	0.006	0.001	0.009	0.080	0.001
Packed-Memory Graph	0.042	0.006	0.008	0.002	0.005	0.063	0.001

Table 21 : Execution times in Luxembourg road network for the 4 different graph representations

Belgium .

Time Performance (sec)							
Graph Representation	**Forward & Backward Tree**	**Combined Tree**	**Plateaux Finding**	**Plateaux evaluation**	**AG construction**	**Total time**	
	average	*average*	*average*	*average*	*average*	*average*	*variance*
Adjacency List	0.919	0.122	0.177	0.041	0.090	1.349	0.549
Forward Star	0.607	0.092	0.213	0.116	0.038	1.067	0.518
Dynamic Forward Star	0.796	0.114	0.184	0.043	0.095	1.232	0.458
Packed-Memory Graph	0.629	0.099	0.218	0.087	0.043	1.076	0.493

Table 22 : Execution times in Belgium road network for the 4 different graph representations

The Netherlands .

Time Performance (sec)							
Graph Representation	**Forward & Backward Tree**	**Combined Tree**	**Plateaux Finding**	**Plateaux evaluation**	**AG construction**	**Total time**	
	average	*average*	*average*	*average*	*average*	*average*	*variance*
Adjacency List	1.612	0.191	0.288	0.043	0.134	2.268	1.198
Forward Star	1.069	0.145	0.347	0.163	0.056	1.780	1.045
Dynamic Forward Star	1.382	0.179	0.304	0.045	0.143	2.053	0.999
Packed-Memory Graph	1.128	0.156	0.354	0.143	0.066	1.847	1.055

Table 23 : Execution times in the Netherlands road network for the 4 different graph representations

Italy .

Time Performance (sec)							
Graph Representation	**Forward & Backward Tree**	**Combined Tree**	**Plateaux Finding**	**Plateaux evaluation**	**AG construction**	**Total time**	
	average	*average*	*average*	*Average*	*average*	*average*	*variance*
Adjacency List	4.656	0.588	1.126	0.140	0.409	6.919	11.813
Forward Star	2.846	0.450	1.380	0.636	0.173	5.486	10.922
Dynamic Forward Star	3.975	0.584	1.082	0.143	0.454	6.238	9.381
Packed-Memory Graph	2.950	0.477	1.380	0.567	0.202	5.575	10.780

Table 24 : Execution times in Italy road network for the 4 different graph representations

Germany .

Time Performance (sec)							
Graph Representation	**Forward & Backward Tree**	**Combined Tree**	**Plateaux Finding**	**Plateaux evaluation**	**AG construction**	**Total time**	
	average	*average*	*average*	*average*	*average*	*average*	*variance*
Adjacency List	9.041	0.970	2.039	0.241	0.684	12.976	64.295
Forward Star	5.820	0.715	2.479	0.735	0.279	10.028	39.155
Dynamic Forward Star	7.684	0.909	2.099	0.250	0.734	11.676	39.632
Packed-Memory Graph	6.053	0.766	2.507	0.949	0.316	10.591	43.215

Table 25 : Execution times in Germany road network for the 4 different graph representations

Remarks .

- The main outcome of this experimental study is that the forward star implementation achieves the best total time performance. In the second place, not far from the forward star, we meet the packed-memory graph, while adjacency list is in the last place. This superiority of the forward star is based on the fact that nodes and edges are stored in consecutive, non-overlapping memory segments that are scanned in maximum efficiency. Thus, processes such as the construction of the forward, backward tree and

the alternative route graph that require sequential accesses of nodes and edges, are done in the least possible time. On the contrary, adjacency list's execution times are inferior since the layout of the graph in the memory is random.

- To be more specific, the packed-memory graph representation is almost 25% faster in query time than the adjacency list. Moreover the forward star representation is almost 3% faster than the packed-memory graph representation. This superiority of the two data structures is due to the fact that, at the expense of a small space overhead, they achieve greater locality of references, less cache misses and hence, better performance in query time.

- Although the total time is best for the forward star representation, as far as the middle stages of the algorithm are concerned, namely plateaux finding and evaluation, adjacency list seems to be better than all the other structures. These two stages does not access all the nodes sequentially, but "chains" of them. The starts of these chains are not necessarily consecutive, thus, a more random layout of the graph in memory may be more helpful.

- The variance of the total time is getting larger as the map size increases. The pairs of source-target are more heterogeneous in larger maps (some of them are close enough, while some can be very remote) and as a result, the time execution may vary enough from query to query, leading to larger variance value.

- Finally, the graph reading mainly for the forward star and the packed-memory graph representation is big enough, since the graph storing in the memory should be done properly using consecutive slots and use of arrays.

Conclusion .

Comparing the experimental results from the above tables, we conclude that the forward star implementation for our static graphs yields the best time performance. It requires, of course, more memory, but the query time is almost 30% better than in the adjacency list. Packed-memory graph and dynamic forward star are also quite good alternatives for our static road networks.

6.2.4 Time performance of plateau method with penalization of edges *(goodness_threshold = 1.0)*

In the following section, we present the time performance of the hybrid method and we make a comparative evaluation with the corresponding execution times of plateau method. The experiments have been done with goodness upper bound equal to 1.0 and with the use of adjacency list representation of the graphs. The results refer to the 1st approach of plateau method with penalization of edges.

Times(sec)								
Map	Method	Forward & Backward Tree	Combined Tree	Plateaux Finding	Plateaux evaluation	AG construction	Total time	Total Time including graph reading & weight calculation	Increase (%)
Luxembourg	plateau	0.063	0.008	0.007	0.001	0.009	0.088	0.743	34.3
	hybrid	0.085	0.008	0.007	0.009	0.009	0.118	0.783	
Belgium	plateau	0.919	0.122	0.177	0.041	0.090	1.349	12.954	39.9
	hybrid	1.264	0.122	0.177	0.235	0.090	1.888	13.509	
Netherlands	plateau	1.612	0.191	0.288	0.043	0.134	2.268	20.314	36.7
	hybrid	2.194	0.192	0.290	0.288	0.135	3.099	21.157	
Italy	plateau	4.656	0.588	1.126	0.140	0.409	6.919	67.384	44.7
	hybrid	6.495	0.589	1.121	1.399	0.410	10.015	70.120	
Germany	plateau	9.041	0.970	2.039	0.241	0.684	12.976	123.000	45.3
	hybrid	12.546	0.977	2.055	2.587	0.687	18.852	125.460	

Table 26 : Time performance of plateau method in comparison to the 1ˢᵗ hybrid approach

Remarks .

- We find out that the execution times of the hybrid method are significantly larger than those of plateau. Specifically, the increase is about 40%. This increase is due to the additional time needed in order to initialize the penalties of the edges and the additional time to penalize the edges of each chosen alternative route plus the edges due to the rejoin penalty applied. Since there is no limit to the number of times each edge can be penalized, some penalizations can be done multiple times leading to great time overheads.

Conclusion .

The 1ˢᵗ approach of hybrid method leads to improvement of the AG quality by about 5% compared to plateau method, while at the same time it leads to a time overhead of 40%. Consequently, we believe that this trade-off is not satisfactory enough and thus, plateau method is better on average.

6.2.5 Summary

The main results of the experiments conducted are presented below in the corresponding categories:

AG and alternative routes quality .

- The 1st approach of the hybrid method turned out to better than plateau method and 2nd hybrid method, for both goodness thresholds by about 5 to 10%.
- The 2nd hybrid method achieved to add slightly better alternatives regarding the goodness criterion.

Memory Usage .

- Adjacency list and forward star representations used the least memory to store the graphs, while packed-memory graph required the most.

Time performance .

- Regarding the time performance depending on the graph data structure, for the same pairs of $s - t$ queries, the forward star representation yields the best execution times.
- Regarding the time performance of the three different methods, plateau method achieves the best results.

Generally, plateau method, either alone or in combination with other methods is a meaningful method which exhibits superior performance compared to the performance of other methods in the field of alternative route planning used as reference from the paper [BDGS11].

Chapter 7

CONCLUSION AND FUTURE WORK

This final chapter gives a short conclusion on the most important aspects of this thesis and reports the most significant results obtained by the experimental studies that have been performed. Also, a perspective for further work in this field of research is presented, trying to suggest possible directions that are worth to be taken in the future.

7.1 Conclusion

The main goal of this thesis is to provide an insight in the field of navigation regarding the aspect of alternative route planning in road networks. Apart from introducing some basic algorithms for alternative routes identification, such as k-shortest paths, disjoint paths, penalty method, our main contribution is to present the notion of admissibility, that is to say, the notion of alternatives that look more natural to humans. We used ways to evaluate alternative routes and metrics to measure the quality of the alternative route graph. Furthermore, we compared methods to compute AGs using plateau method and a combination of plateau method with penalty. Our experiments showed that the combination of plateau with penalty achieves the best quality for the AGs. However, plateau method has much better time performance. Moreover we examined the time performance within the scope of different graph data structures and we concluded that for our static real-world maps, forward star representation achieves the best results. To sum up, plateau method, either alone or in combination with other methods, is a meaningful method which exhibits superior performance compared to the performance of other methods in the field of alternative route planning used as reference from the paper [BDGS11].

7.2 Outlook

The plateau method we implemented has a number of interesting directions for future work.

Weight function . At this level, the edges weights are calculated once, in the beginning of the execution, are static and it is not possible to change without re-reading the graph. So, the cost function that is used to compute the weights of the graph can incorporate any of the known factors. It may be sensitive to both time and duration, can incorporate real-time of historical

traffic information, can take into account user preferences and can use financial information such as road pricing.

Real-time Updates . Currently, our plateau implementation only supports static road networks. An improvement would be to implement dynamization techniques so as to incorporate node or edges updates that are caused, for example, by the opening of a road. These dynamizations techniques would be able to update the graph without re-reading the map from scratch.

Speed-up Techniques . The basic time overhead in our implementation is introduced by the two Dijkstras running to form the forward and backward trees. Thus, it would be desirable to use a variation of Dijkstra's algorithm so as to improve the time performance of this stage. These variations use special heuristics to explore the nodes or edges in a better order, or to terminate earlier, or to explore major roads only, or to explore outwards from source and destination simultaneously. They do this in order to run faster or to use less memory. It should be understood that plateau method of combining forward and backward trees to find plateaux is independent of how these trees were computed. So, if these variations, known as speed-up techniques, are applied, then plateau method can yield diverse alternative routes needing less running time or memory use.

Metrics . Currently, we evaluate plateaux using totalDistance, averageDistance, decisionEdges, based on an objective function. But once we have reduced the number of plateaux to the most interesting ones by using goodness value, we can filter them in many other ways. For example, we may choose to order them based upon user preferences, such as motorways, fewer junctions, lower tolls, driving costs, and familiarity.

APPENDIX

Appendix – Technical details and Implementation's interface

In this chapter of the appendix, we give some technical details useful for everyone that wishes to use our C++ programs, set up the environment and execute them. Obviously, it is necessary to have a C++ compiler in order to execute the programs. Note that we used the g++ compiler in a linux environment, so the instructions are given, provided that the setup is going to take place in a similar environment (see 6.1.1).

First of all, both plateau and hybrid methods are separated into multiple header files which are necessary for the proper functionality of the program. If any of the files needed is missing, the program cannot be compiled and as a result executed. In these separate files, various C++ classes are contained, implementing the different stages of the algorithm, such as Dijkstra, BackwardDijkstra, Plateau etc. as well as the various data structures used, such as priority queues, graph representations etc. Note that these C++ files are part of "pgl" library which implements many algorithms and data structures in the wide field of route planning. Besides that, for a user that wishes to execute the code, it is necessary that the various header files are in the proper directories, as these are stated in the include statements found in the first few lines of every file. Our implementations, apart from the files that already exist in pgl, are given in the following files : plateau.cpp (including main function), plateau.h, subgraph.h and a Makefile for plateau method and in files : plateau_penalty.cpp (including main function), penalty.h, subgraph.h and a Makefile for the hybrid method. Makefiles should be in the same directories with plateau.cpp and in plateau_penalty.cpp respectively. Apart from them, in order to compile the program, it is necessary to install boost library mainly for "random" and "program options" libraries.

If no file is missing and everything is well suited in the right directories, the executable file is produced by executing the command "make" (being in the directory where Makefile is found). The executable a.out (as denoted in our Makefiles) can be now executed and give results. A user is able to run the program without having any further knowledge on the code. It is sufficient to execute the program, as follows, and simply gather the results. So, by running the executable, namely "./a.out", one can see the line arguments needed. In our implementations, a typical execution would be :

$$./a.out - s\ 1000 - p\ 11 - g\ 0 - f\ 0 - m\ belgium$$

Where

$-s$ [--size]   number of queries to execute. Default : 1

97

$-\boldsymbol{p}$ [--numP]   max number of alternatives to keep. Values>=1. Default : 1 corresponds to the shortest path. Default = 11

$-\boldsymbol{g}$ [--graphtype]   graph type. Adjacency List[0], PackedMemoryArray[1], Forward Star[2], Dynamic Forward Star [3]. Default : 0

$-\boldsymbol{f}$ [--format]   map format. DIMACS10[0]. Default:0

$-\boldsymbol{m}$ [--map]   input map. The name of the map to read. Maps should reside in '$HOME/Projects/pgl/Graphs/DIMACS10/' and should consist of 2 files, both with the same map name prefix, and suffixes 'osm.graph' and 'osm.xyz'. Default:'luxembourg'

The aforementioned arguments list describes the use of each argument, and the possible values for execution. Note that the execution succeeds if the necessary map files are downloaded (possibly from http://www.cc.gatech.edu/dimacs10/archive/streets.shtml) in the directory "$HOME/Projects/pgl/Graphs/DIMACS10/". Note also that the pairs of $s-t$ are chosen randomly by a random generator.

The output of the program, either referring to AG and alternative routes quality or memory usage and time performance (depending on which of the two cases is uncommented) is given in a file denoted by the name of the in question map + ".txt" (i.e. luxembourg.txt).

For users that want to go a bit further, that is to say, to understand to a greater extent the structure of the code and possibly change some upper bounds (i.e. δ, goodness_threshold, averageDistance upper bound etc.), we present the prototypes of the basic methods executed including their arguments.

The basic method called from main function (in both implementations) is the :

$$runExperimentsAt(G, subG, s, t, numPlateaux, outfile)$$

which takes as input, the graph G (or fsG, $pmaG$ etc), the alternative route $subG$, the source and the target nodes, s and t respectively, the maximum number $numPlateaux$ of alternatives to compute and a file to output the results.

This method contains the computation of forward, backward, combined Trees, finds plateau paths and creates alternative subgraph (with proper function calls).

In more detail, it calls :
- runQueries<Dijkstra<GraphType> >(G, s, t);
- runQueries<BackwardDijkstra<GraphType> >(G, s, t);

The runQuery method is implemented with use of templates so as to be called with Dijkstra template argument the first time and BackwardDijkstra the second. The arguments are the graph G and the source and target nodes s, t.

Then, it instantiates a Plateau (or Penalty) variable called plateauFinder,

- Plateau<GraphType> plateauFinder(G);

and calls

- plateauFinder.buildCombinedTree(s);
  Source node s is given as argument.
- plateauFinder.findPlateaux(0.85,s,t);
  The argument list includes the goodness_threshold which now is set to 0.85 and the source and target nodes.
- plateauFinder.evaluatePlateaux(nump,s,t,outfile);
  The arguments list includes the maximum number of alternatives to be computed, given as input to the program (now set to 0.85), the source s and target t nodes and a reference to the output file.

Finally, it instantiates a Subgraph variable,

- Subgraph< GraphType, subGraph> AG(G, subG);

And then it calls

- AG.createAlternativeGraph();

Note that change in the thresholds of δ, averageDistance can be done manually in the header files "plateau.h" or "penalty.h".

Bibliography

[ADGW10]   Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, Alternative Routes in Road Networks, SEA2010.

[AEB00]   V. Akgun, E. Erkut, R. Batta, On finding dissimilar paths, European Journal of Operational Research 121 (2000) 232-246.

[Bau11]   Moritz Baum, On Preprocessing the Arc-Flags Algorithm, Master Thesis, Department of Informatics, Institute of Theoretical Computer Science, Karlsruhe Institute of Technology (KIT), September 2011.

[BDGS11]   R. Bader, J. Dees, R. Geisberger, P. Sanders, Alternative Route Graphs in Road Networks, TAPAS 2011

[Bel58]   Richard Bellman, On a routing problem, Quarterly of Applied Mathematics, 16(1):87–90, 1958.

[Bha94]   Bhandari R., Optimal diverse routing in telecommunication fiber networks, Proceedings of IEEE INFOCOM '94, Toronto, Ontario, Canada, vol. 3, June, 1994; 1498–1508.

[BK60]   Bellman, R. AND Kalaba, R. "On kth Best Policies," J. of SIAM, Vol. 8, No. 4 (December 1960), pp. 582-588.

[BKH57]   Bock F., Kantner H. and Haynes J., An Algorithm for Finding and Ranking Paths Through a Network, Research Report, Armour Research Foundation, Chicago, Illinois, November 15, 1957.

[CKR63]   Clarke S., Krikorian A. and Rausan J., "Computing the N Best Loopless Paths in a Network," J. of SIAM, Vol. 11, No. 4 (December 1963), pp. 1096-1102.

[Del09]   Daniel Delling, Engineering and Augmenting Route Planning Algorithms, Department of Informatics, University of Karlsruhe (TH), February 2009

[DGJ09]   Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, Shortest Paths: Ninth DIMACS Implementation Challenge, DIMACS Book, American Mathematical Society, 2009.

[Die97]   Diestel R., Graph Theory, Graduate Texts in Mathematics, Springer, New York, 1997.

[Dij59]   Edsger W. Dijkstra, A Note on Two Problems in Connexion with Graphs, Numerische Mathematik, 1:269–271, 1959.

[DW09]   Daniel Delling, Dorothea Wagner, Pareto Paths with SHARC, SEA 2009.

[Epp94]   Eppstein David, Finding the k Shortest Paths, University of California, May 31 1994.

[FF62]   Ford LR, Fulkerson DR. ,   Flows in Networks, Princeton University Press, Princeton, NJ, 1962.

[Fuc10]   Fabian Fuchs, On preprocessing of the ALT-Algorithm, Student thesis, Faculty of Computer Science, Institut for Theoretical Informatics(ITI), Karlsruhe Institute of Technology (KIT), 2010.

[Gei08]   Robert Geisberger, Contraction Hierarchies, Master thesis, Universitat Karlsruhe (TH), Fakultat fur Informatik, 2008.

[GH05]   Andrew V. Goldberg and Chris Harrelson, Computing the Shortest Path: A* Search Meets Graph Theory. In Proceedings of the 16th Annual ACM– SIAM Symposium on Discrete Algorithms (SODA'05), pages 156–165, 2005.

[GKM03]   Yuchun Guo, Fernando Kuipers and Piet Van Mieghem, Link-disjoint paths for reliable QoS routing, International Journal of Communication Systems, Int. J. Commun. Syst. 2003; 16:779–798 (DOI: 10.1002/dac.612), 2003.

[GKW06a]   Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06), pages 129-143. SIAM, 2006.

[GSSD08]   Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08), volume 5038 of Lecture Notes in Computer Science, pages 319–333. Springer, June 2008.

[Gut04]   Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), pages 100111. SIAM, 2004.

[GW05]   Andrew V. Goldberg and Renato F. Werneck, Computing Point-to-Point Shortest Paths from External Memory. In Proceedings of the 7thWorkshop on Algorithm Engineering and Experiments (ALENEX'05), pages 26–40. SIAM, 2005.

[Han79]   Hansen P., Bicriteria Path Problems. In Fandel G., Gal T., eds.: Multiple Criteria Decision Making – Theory and Application –. Springer (1979) 109–127.

[HKMS06]   Moritz Hilger, Ekkehard Kohler, Rolf H. Mohring, and Heiko Schilling, Fast point-to-Point Shortest Path Computations with Arc-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, 9th DIMACS Implementation Challenge - Shortest Paths, November 2006.

[HNR68]   Peter E. Hart, Nils Nilsson, and Bertram Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics, 4:100–107, 1968.

[HP59]   Hoffnam W. and Pavley R., "A Method for the Solution of the Nth Best Problem," J. of ACM, Vol. 6, No. 4 (October 1959), pp. 506-514.

[Jon12]   Alan Henry Jones, Method of and apparatus for generating paths, United States Patent, No. US 8,249,810 B2, August 21,2012.

[Jr.56]   L.R. Ford Jr., Network flow theory, Paper P-923, The RAND Corporation, Santa Monica, California, August 1956.

[Kar07]   George Karypis, METIS - Family of Multilevel Partitioning Algorithms, Digital Technology Center, University of Minnesota, 2007.

[Lau97]   Ulrich Lauther, Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[Lau04]   Ulrich Lauther, An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In Geoinformation und Mobilitat - von der Forschung zur praktischen Anwendung, volume 22, pages 219–230. IfGI prints, 2004.

[Mar84]   Martins, E.Q., On a Multicriteria Shortest Path Problem, European Journal of Operational Research 26(3), 1984 236–245.

[MMPZ12]   Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, Christos Zaroliagis, A New Dynamic Graph Structure for Large-Scale Transportation Networks, eCOMPASS European Project, Project Number 288094, October 2012.

[MMZ12]   Georgia Mali, Panagiotis Michail, Christos Zaroliagis, A New Dynamic Graph Structure for Large-Scale Transportation Networks, eCOMPASS European Project, July 2012.

[MS04]   Burkhard Monien and Stefan Schamberger, Graph Partitioning with the Party Library: Helpful-Sets in Practice. In Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBACPAD' 04), pages 198–205. IEEE Computer Society, 2004.

[MSS+06]   Rolf H. Mohring, Heiko Schilling, Birk Schutz, Dorothea Wagner, and Thomas Willhalm, Partitioning Graphs to Speedup Dijkstra's Algorithm, ACM Journal of Experimental Algorithmics, 11:2.8, 2006.

[Paj09]   Thomas Pajor, Multi-Modal Route Planning, Diploma thesis, Department of Informatics, Institute of Theoretical Computer Science, Karlsruhe Institute of Technology (KIT), March 2009.

[Pel07]   Francois Pellegrini, SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.

[Pol61]   Pollack, Solutions of the k Best Route Through a Network-A Review, J. of Math. Anal. And Appl., Vol. 13, No. 3 (August-December 1961), pp. 547-559.

[PS82]   Papadimitriou CH, Steiglitz K., Combinatorial Optimization–Algorithms and complexity. Prentice–Hall, Inc.: Englewood Cliffs, NJ, 1982.

[Sak66]   Sakarovitch, The k Shortest Routes and the k Shortest Chains in a Graph, Opns. Res. Center, University of California, Berkeley, Report ORC-32, October 1966.

[Sch08]   Dennis Schieferdecker, Systematic Combination of Speed-Up Techniques for Exact Shortest-Path Queries, Diploma thesis, Department of Informatics, University of Karlsruhe, January 2008.

[SS05]   Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05), volume 3669 of Lecture Notes in Computer Science, pages 568–579. Springer, 2005.

[SS06a]   Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06), volume 4168 of Lecture Notes in Computer Science, pages 804–816. Springer, 2006.

[ST84]   Suurballe JW, Tarjan RE. A quick method for finding shortest pairs of disjoint paths, Networks 14:325–333, 1984.

[Suur74]   Suurballe JW., Disjoint paths in a network, Networks 14: 125–145, 1974.

[SWW99]   Frank Schulz, Dorothea Wagner, and Karsten Weihe, Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99), volume 1668 of Lecture Notes in Computer Science, pages 110 - 123. Springer, 1999.

[WW03]   Dorothea Wagner and Thomas Willhalm, Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03), volume 2832 of Lecture Notes in Computer Science, pages 776–787. Springer, 2003. (Cited on page 2.)

[WWZ05]   Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis, Geometric Containers for Efficient Shortest-Path Computation. ACM Journal of Experimental Algorithmics, 10:1.3, 2005. (Cited on page 2.)

[Yen71]   Jin Y. Yen, Finding the K Shortest Loopless Paths in a Network, University of Santa Clara, 1971, pp.712-716.